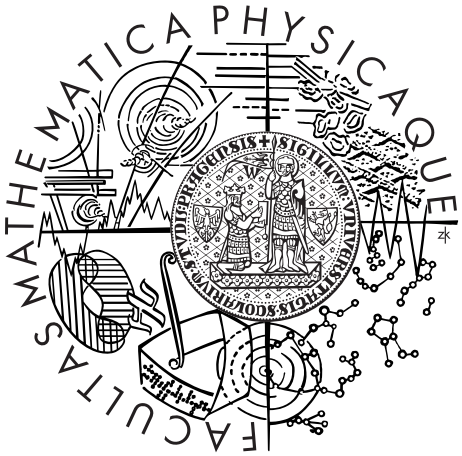
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Bc. Ľuboš Magic

Remote Method Invocation for Android Platform

Department of Distributed and Dependable Systems

Supervisor: RNDr. Tomáš Pop
Study programme: Computer Science
Specialization: Software Systems

Prague 2012

The author would like to thank Tomáš Pop, Jan Šteffl and Jan Nemec for their interest and support of this work.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ..... date ..... Ľuboš Magic

Názov práce: Vzdialené volanie metód pre platformu Android

Autor: Bc. Ľuboš Magic

Katedra: Katedra distribuovaných a spoľahlivých systémov

Vedouci diplomovej práce: RNDr. Tomáš Pop, Katedra distribuovaných a spoľahlivých systémov

Abstrakt: Diplomová práca skúma potenciál vzdialeného volania metód v prostredí mobilných zariadení s operačným systémom Android. Hlavný cieľ práce je výskum v oblasti realizácie bezpečnostne kritických častí aplikačného kódu na smart karte (v prostredí mobilných zariadení hovoríme o SIM karte). Práca taktiež popisuje vzdialené volanie metód všeobecne a porovnáva špecifiká implementácie volaní na vzdialený server a na smart kartu. Súčasťou práce je aj návrh prípadovej štúdie, ktorá využíva získané poznatky.

Kľúčové slová: Android, vzdialené volanie metód, smart karta, bezpečnosť

Title: Remote Method Invocation for Android Platform

Author: Bc. Ľuboš Magic

Department: Department of Distributed and Dependable Systems

Supervisor: RNDr. Tomáš Pop, Department of Distributed and Dependable Systems

Abstract: The thesis inquires the potential of a remote method invocation in the context of the Android mobile devices. The primary goal of the thesis is to investigate execution of security-critical parts of application code on smart cards (a prominent example of a smart card is the SIM card). Further, the thesis discusses issues of implementation of the remote method invocation in general, covering also its other frequent forms (such as execution on a remote server). A part of the thesis is a real world case study, which demonstrates the results of the thesis.

Keywords: Android, remote method invocation, smart card, security

Table of Contents

Chapter 1

Introduction

The first commercial version of the Google's Operating system, Android 1.0, was introduced on 23 September 2008 [1]. Soon after the release, on 22 October 2008, the first Android phone "HTC Dream" entered the US Market [2]. Instantly after the debut of the first mobile device with Android, the market-share of this operating system started to grow rapidly. The Table 1.1 shows the Mobile phone operating systems' market-share in time according to Gartner [3].

Year	Symbian	Android	RIM	iOS	Microsoft	Bada	Others
2011 Q2	22.1%	43.4%	11.7%	18.2%	1.6%	1.9%	1.1%
2011 Q1	27.4%	36.0%	12.9%	16.8%	3.6%	N/A	3.3%
2010	37.6%	22.7%	16.0%	15.7%	4.2%	0.9%	2.9%
2009	46.9%	3.9%	19.9%	14.4%	8.7%	N/A	6.1%
2008	52.4%	0.5%	16.6%	8.2%	11.8%	N/A	10.5%
2007	63.5%	N/A	9.6%	2.7%	12.0%	N/A	12.1%

Table 1.1: Mobile phone operating systems' market-share in time.

Plenty of applications need to securely store a secret. The number of such an Android applications is rising [4] and in-hand with the Android penetration, the need for a secure storage will become more and more relevant. The secure storage shall not be implemented in a phone's memory, as there exist a risk of secret compromission by a malware application (especially when Android system can contain various serious security vulnerabilities [5]). A better place for storing the secret is a hardware token specially designed for security sensitive operations, the SIM card. Naturally, the SIM card cannot just store the secret and reveal it when a password is provided, as that would lead to the same security risk as storing the secret outside of the secure token. Instead, the whole computation with the secret, must be held inside the secure token and only the result of such a computation shall be revealed.

The Java Mobile Edition enabled mobile phones are able to access the Smart card interface by a technology called JSR177, Security and Trust Services API for J2ME. This technology defines the methods to communicate to a Smart Card,

by leveraging the APDU protocol to the J2ME application interface [6]. However, there is no standardized API for Android phones and the Smart cards yet, although some initiatives to create one already exist [7].

1.1 Motivation Case Study

In this section, we present a motivation case study to show the potential benefits of the remote method invocation on smart cards in the Android environment.

As the computational power of computers is constantly rising, the brute force attacks for static passwords are becoming more and more successful. As the human being has its mental limitations of remembering a password of a certain complexity, the two-factor authentication was introduced to improve the security of authentication schemes. There are several ways how to implement the two-factor authentication. Most of them rely on a OTP (One Time Password) delivered to the user by various channels, such an SMS, e-mail, IM, and others. The security flaw in these is obvious, the security of the solution depends on the security of the channel used. The other way of presenting the user the OTP synchronized with the authentication server, bypassing the possibly insecure transfer channel, is the hardware token in the user's possession as shown in Figure 1.1.

Figure 1.1: Hardware token generating OTP [8].

The drawback of this method is the high cost of the hardware token if used with a large user base and the low level of user friendliness, when the user must carry the device all the time. By implementing the remote method invocation between the Android applications and the smart cards, we would allow a solution, which would use a SIM card as a secure token and would present the OTP to the user on the Android phone, to be created. The JCRE (Java Card Runtime Environment) embedded in a SIM card would run the algorithm and would use the pre-shared secret, which would be stored in the SIM card's file system as well. The access to a SIM card's file system would have to be restricted.

The solution shall have a clear interface, which could be used by i.e. a bank

application to allow the user accessing his/hers e-banking services with a transparent second factor authentication to the user.

There are two well-known standardized One Time Password algorithms, published by IETF (The Internet Engineering Task Force), HOTP (Hashed Message Authentication Code OTP) [10] and TOTP (Time-based OTP) [12], which we discuss more in detail in Chapter 3. The two mentioned algorithms require a pre-shared secret to be used, which would not be safe to keep inside an Android device memory, but a SIM card tamper resistant environment appears to be an ideal storage. Both of the algorithms may be implemented into a scheme and serve as an authentication system for Internet enabled applications user policies.

1.2 Goals

The thesis will inquire the potential of a remote method invocation in the context of the Android mobile devices. The primary goal of the thesis is to investigate execution of security-critical parts of application code on smart cards. As a result of the investigation, the real-world case study will be proposed and implemented.

1.3 Structure of the Thesis

Chapter 2 offers an insight into a related work. The technical background of the thesis is described in Chapter 3. In Chapter 4 we provide a detailed analysis of what has to be done to fulfill the goals of the thesis and we provide a reasoning of the chosen approaches. Chapter 5 describes the implementation of the proposed real-world case study and the underlying RMI environment. Chapter 6 discusses important aspects of the proposed and implemented solution and it also elaborates on various interesting topics related to this thesis. We provide a conclusion and a vision of future work in Chapter 7.

Chapter 2

Related work

The two most important projects related to this work are the Security and Trust Services API for J2ME platform and the Secure Element Evaluation Kit for the Android platform.

2.1 Security and Trust Services API for J2ME

The Security and Trust Services API specification defines optional packages for the Java Micro Edition Platform [6]. It has been produced in response to Java Specification Request 177 and specifies a collection of APIs that provides security and trust services by integrating a Security Element access. One of the form of the Security Element is also a SIM card. The scope of the *SATSA* specification deals with several needs, which can be divided into the following groups:

- Smart Card Communication - The specification defines two smart card access methods. The first one is based on APDU (Application protocol data unit) protocol and the second one uses Java Card RMI protocol. These two access methods allowed a J2ME application to communicate with a smart card by leveraging the security services deployed on it.
- Digital Signature Service and Basic User Credential Management - Digital signature service and credential management rely on a Security Element to provide secure storage of user credentials and cryptographic keys, as well as to perform secure computation involving the cryptographic keys that are securely stored on the Security Element. The API defined by *SATSA* reduces the complexity of generating a formatted digital signature by introducing this high-level interface to J2ME applications.
- General Purpose Cryptography Library - It is a subset of the Java Standard Edition Platform cryptography API (version 1.4.2), which supports basic cryptographic operations, such as message digest, digital signature verification (but not signing), encryption, and decryption.

The technology itself did not enjoyed a wide spread usage, as only a few handset manufacturers included a full support in their devices. However, the API specification helps us to identify the needs, which we may target our implementation of the Remote Method Invocation for Android platform on.

2.2 Secure Element Evaluation Kit for the Android platform

An international security company called Giesecke & Devrient[13] identified a need of a secure token presence in an open environment of the Android platform. The company offers a flash memory MicroSD card[14] with an embedded smart card chip and they have required an access to it from an Android application. A rising demand for NFC (Near Field Communication) payment applications in Europe and a rising number of NFC enabled handsets with Android operating system available, the business need for an Android system and a secure token interface became even more eminent.

The NFC communication is possible on a physical layer with no interaction of the Android system, but it is often desired, that the payment terminal - secure token communication can interact with an Android application. The application can either allow/dismiss the payment, show a summary of the payment or prompts a user to authorize the payment.

For those reasons, the company started an initiative[15] for creation of a smart card API between a smart card and the Android applications, which would be consequently included in the Android baseline and thus available in new handsets.

At the time, when this thesis has been started, the mentioned initiative was not publicly known. The public availability of the initiative, which partly helps to fulfill the goals of the thesis, came shortly after the thesis announcement.

The initiative proposes a vision of how an access to a secure element in the Android environment should look like. The vision of this initiative is an APDU interface to the smart card, which would be accessible by an Android application. The initiative copes with a secure element access security only marginally, it depends on the security of the secure token itself.

The activity on the initiative is very high as several people are regularly contributing to it. We follow this initiative and we will possibly reuse some ideas or components of the initiative in the implementation phase of the thesis, depending on how the initiative evolves.

Chapter 3

Background

This chapter describes technical background of the thesis. It focuses on the most important aspects of the technologies used or related to the thesis. It does not aspire to be an exhaustive manual of all the possibilities that the technologies may offer. The main purpose is to help a reader to introduce the fundamentals, which are used in the implementation of the thesis goals. The simplifications have been made in order to keep the scope in an acceptable scale, but interested reader has the possibility to read all the details in referenced documents. First section describes the basics of the Android Operating system, the second section introduces basic principles of Smart Cards and the last section familiarizes the reader with a general concept of the Remote Method Invocation.

3.1 Android Operating System

Android is a software stack for mobile devices that includes an operating system, middleware and key applications [16].

The Android system is developed by an Open Handset Alliance, a consortium of a hardware, software and telecommunication companies, which have an interest in advancing open standards for mobile devices [17]. The source code of the whole project was made public under the Apache license and is maintained by the Android Open Source Project community, led by Google [18].

Android architecture is hierarchically structured as shown in Figure3.1. The framework for applications on Android is extremely rich in comparison to older mobile platforms, such as Nokia's Symbian. It allows developers of the applications to use the device hardware, access location information, run background services, set alarms, add notifications to the status bar, and more. Developers have full access to the same framework APIs as are used by the core applications, as an Internet browser, SMS program, calendar, etc. The application architecture allows the reuse of components as any application can publish its capabilities and any other application may use it. Applications for the Android are written in Java programming language. Application Java source code is compiled into the bytecode *.class* format and then transformed into the *.dex* format by a tool which comes with the Android SDK. The created classes are then run by a Dalvik virtual machine [19]. The Dalvik VM is a Google's implementation of a standard Java runtime. The compact Dalvik Executable format is designed to be suitable for systems that are constrained in terms of memory and processor speed. Every

Android application runs in its own process, with its own instance of the Dalvik virtual machine. The Dalvik VM relies on the Linux kernel for underlying functionality such as threading and low-level memory management. The Android is built on Linux kernel version 2.6 for core system services such as security, memory management, process management, network stack, and a driver model. The kernel also acts as an abstraction layer between the hardware and the rest of the software stack [20].

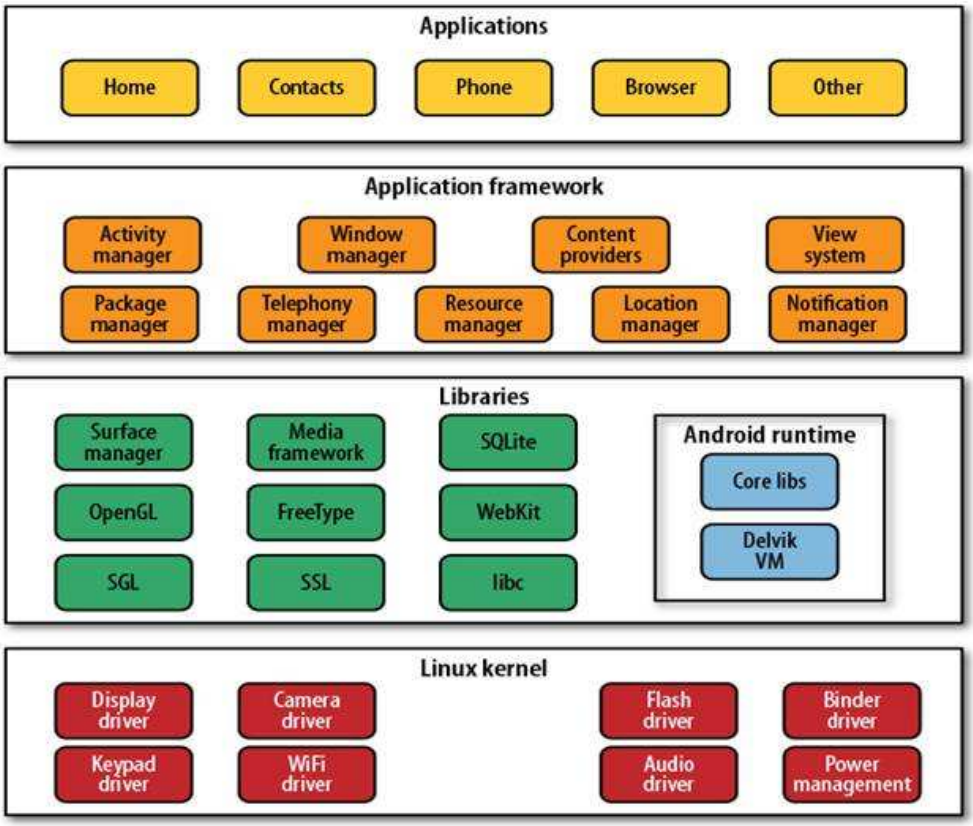


Figure 3.1: System architecture of the Android OS [16].

3.1.1 Application Fundamentals

An Android package is an archive file with an *.apk* suffix which contains all the data and resources files for an application. To get an Android package, we use the Android SDK tools to compile all the resources. To install the application to an Android powered device, only single *.apk* archive is required.

A typical Android application consists of several components. Every component may behave as a different point through which the system can enter the application. There are four different types of an application component in the Android [22]:

- Activities represent a single screen with an user interface. The life-cycle of an activity is shown in Figure 3.2.
- Services are the background tasks that perform long-running operations or work for remote processes. A service does not provide a user interface.

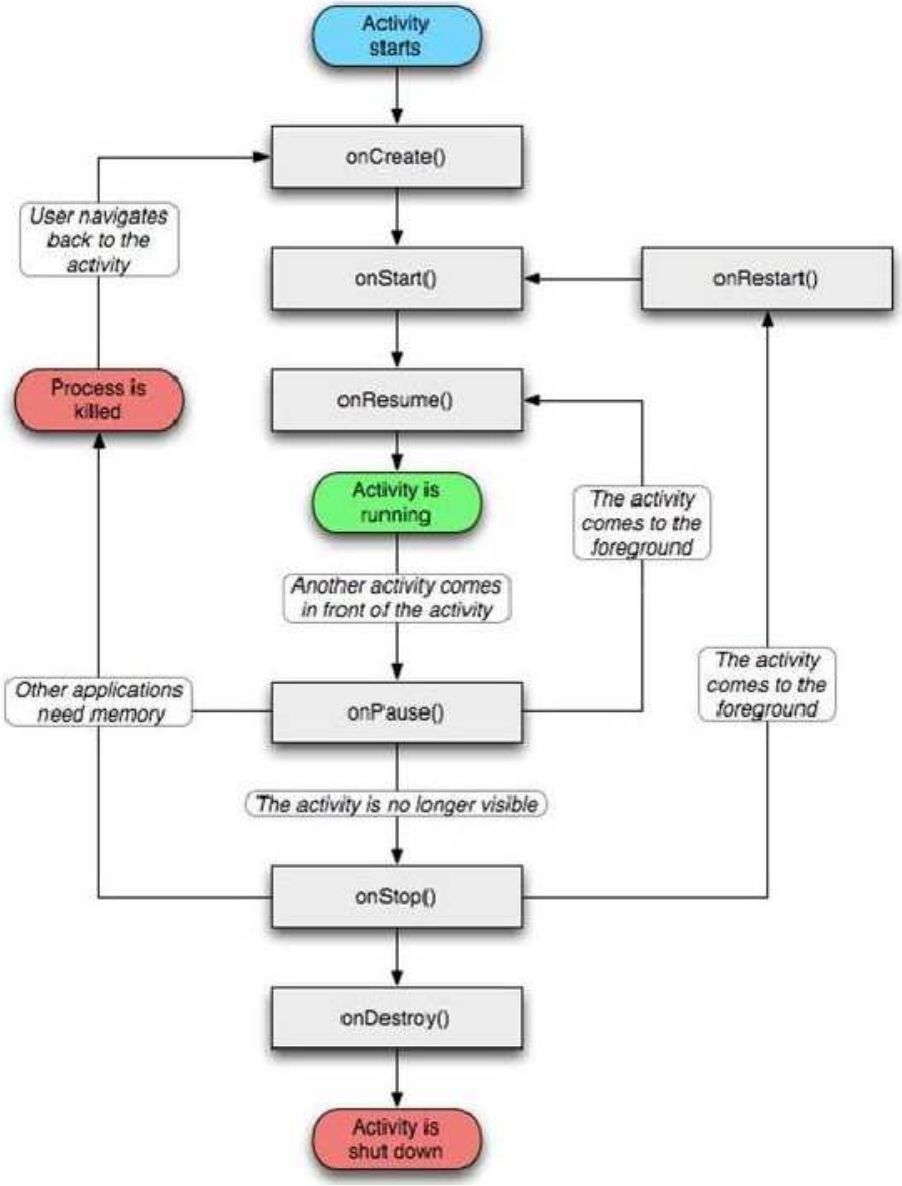


Figure 3.2: Android Activity life cycle [24].

Services may be started by another component, such as an activity, or it may be binded to in order to interact with it.

- Content providers manage shared set of application data. The storage may be either a file system, an SQLite database, a remote location on the web or any other persistent location, which is accessible by the application. The Content provider is used by the applications to query or to modify the persistent data, if the component allows it.
- Broadcast receivers respond to a system-wide broadcast messages. The messages may originate from the Android itself, or even the applications may issue their own announcements. The Broadcast receiver does not show a user interface, but it may notify a user by a status bar notification.

The component system allows a high amount of possible code reuse as An-

droid allows applications to start another application's components. Thanks to this approach, applications do not need to implement activities, which were already implemented in another applications, such as activity for taking picture from a device's camera. When one application calls a component from another application, a new process for that application is started (if it is not already running). However, due to the fact, that Android runs each application in a separate process with file permissions that restrict access to other applications, the application cannot directly activate the other application component, it needs to deliver a message to the system, called *Intent*, to start a particular component. The Android system then evaluates if it is possible and starts the component.

The *Intent* is an asynchronous message which can be used to activate three out of four component types, activities, services and broadcast receivers. Intents bind different components to each other at runtime, regardless whether the owner of the component is the calling application, or another.

3.1.2 The Android Application Manifest File

Every Android application must have an XML (Extensible Markup Language) structured Manifest file, which informs the Android system about the presence of the application components. Without this file, the Android system would not be aware of the application, and as a consequence, it would not be possible to run it.

In addition to declaring the application's components, the Manifest file is also used to identify the required permissions by the application, minimum version of the API supported, hardware features used or required, additional libraries which need to be linked against, and more.

3.1.3 Security Architecture

An essential security architecture building block for Android is the approach of minimal permissions granted to the applications, meaning that by default, an application has no permissions to perform any operations that would adversely impact other applications, the operating system, or the user [21].

After the installation, each application lives in its own sandbox [22]. The Android operating system is a multi-user Linux based system, in which each application acts as a different user. A unique Linux user ID is assigned to each application at install time. All the files contained in an application have the permissions set, so that only the user ID assigned to that application may access them. The kernel enforces security between applications and the system at the process level through standard Linux facilities, such as user and group IDs that are assigned to applications.

Each process in the Android has its own virtual machine, which means, that every application's code is run isolated from others. Each application is run in its own process. The process is started, when a component from the application needs to be executed and shut-down, when it is no longer needed.

Since the application sandbox is in the kernel, this security model extends to native code and to operating system applications as well and is not bound only to the Dalvik VM barriers. All of the software above the kernel (see 3.1), including

operating system libraries, application framework, application runtime, and all applications run within the application sandbox [23] too.

Because of the sandboxes, applications must explicitly share resources and data. This is done by the declaration of the required *permissions*, which are needed for its tasks and are not provided by the basic sandbox. The set of required permissions is declared by the programmer in the mentioned Manifest file. The Android system prompts the user for approval of the application required permissions at install time. There is no mechanisms in the Android, for granting the permissions at run-time, as the architects of the system, considered this possibility as too much user experience disrupting.

Permission Types

An application can declare a security permission that can be used to limit access to specific components or features of this or other applications [27].

The declaration is done via a Manifest file and can have various attributes. Along with the expected ones as the description and labels are, a protection level is also being defined here. It characterizes the potential risk implied in the permission and indicates the procedure the system should follow when determining whether or not to grant the permission to an application requesting it. The value can be set to one of the following:

- *normal* - The default value lower-risk permission that gives requesting applications access to isolated application-level features, with minimal risk to other applications, the system, or the user. The user has an option to review these permissions before installing, but if not explicitly requested, the Android automatically grants this type of permission to a requesting application automatically at install time, without asking for explicit approval.

- *dangerous* - A higher-risk permission that would give a requesting application access to private user data or control over the device that can negatively impact the user. Because this type of permission introduces potential risk, the system will not automatically grant it to the requesting application. Any dangerous permissions requested by an application will be displayed to the user and require confirmation before proceeding, or some other approach may be taken to avoid the user automatically allowing the use of such facilities.

- *signature* - A permission that the system grants only if the requesting application is signed with the same certificate as the application that declared the permission. If the certificates match, the system automatically grants the permission without notifying the user or asking for the user's explicit approval.

- *signatureOrSystem* - A permission that the system grants only to applications that are in the Android system image or that are signed with the same certificates as those in the system image. It is used for certain special situations where multiple vendors have applications built into a system image and need to share specific features explicitly, because they are being built together. It is not intended to be used by a common application.

File System

As every application runs as a unique user, following the standard Linux-based file system permission rules ensures, that one user cannot read nor alter another user's files, unless the files were explicitly exposed by a developer.

Android contains several partitions with various file systems. The */system* partition contains Android's kernel as well as the operating system libraries, application runtime, application framework, and applications. This partition is set to read-only. When a user boots the device into a Safe Mode, only core Android applications are available. This ensures that the user can boot their phone into an environment that is free of third-party software.

The type of a file system used for a certain partition depends mainly on the hardware manufacturer and a version of the OS. A typical file system type for */system* and */data* is YAFFS(2) (Yet Another Flash Filesystem), the */sdcard* partition uses VFAT (Virtual File Allocation Table). An example of a custom file system can be Samsung's proprietary RFS (Robust File System) used in the Galaxy S series handsets [25].

Application Signing

Every Android application must be signed with a certificate prior to its installation. The certificate's private key is held by the application developer. It is not required for the certificates to be signed by a certification authority, the common model is, that an Android application uses self-signed certificates. The main purpose of the certificates in Android is to reliably distinguish applications authors. This allows the system to grant or deny applications access to signature-level permissions and to grant or deny an application's request to be given the same Linux identity as another application, depending whether they were signed by the same key-set. The certificates also allow the developers to issue upgrades of their applications, which are subsequently identified to be the same application only using a different version [26]. Another purpose of the application signing is a confirmation for the users, that the code has not been altered or corrupted since it was signed by use of a cryptographic hash.

3.1.4 Inter-Process Communication

The Android platform is intended to eliminate the duplication of functionality in different applications, to allow functionality to be discovered and invoked on the fly, and to let users replace applications with others that offer similar functionality [28]. In order to achieve this, an inter-component or inter-process communications had to be introduced.

An Andoird application can use the traditional Linux IPC mechanisms as sockets, pipes, message queues, shared memory or others to exchange data with other applications [29] [30] [31]. However, these practices are hard to maintain and easily error prone. Modern programming environments including Android, have moved on to more robust component-like systems. Specifically:

- *Intents* - Representations for operations to be performed. An intent is a kind of a data-structure which contains an URI (Uniform Resource Identifier)

and an action. The URI uniquely identifies an application component and the action identifies the operation to be executed [32].

- *Remote methods* - With the help of AIDL (Android Interface Definition Language) remote objects executed in another process allow to call it's methods through defined API.

3.2 Smart Cards Overview

A smart card may come in one of two varieties. Memory only smart card and a smart card with integrated microprocessor. Memory cards simply store data and can be viewed as a small flash disk with optional security. A microprocessor card, on the other hand, can add, delete and manipulate information in its memory on the card. Similar to a miniature computer, a microprocessor card has an input/output port operating system and hard disk with built-in security features.

The input/output interface of a smart card may be contact, but also contactless. Contact smart cards are inserted into a smart card reader, making physical contact with the reader. However, contactless smart cards have an antenna embedded inside the card (see Figure 3.3) that enables communication with the reader without physical contact. Also cards with both of the input/output type of the interfaces exist [34].

Figure 3.3: Smart card with contactless interface [33].

Smart cards are used in a wide range of businesses because of its ability to provide security for transactions for relatively low investment and maintenance cost [35]. The main areas of usage of smart cards are:

- Telecommunications - The most prominent application of smart card technology is in SIMs (Subscriber Identity Modules), required for all phone systems under the GSM (Global System for Mobile Communication) standard.

Each phone utilizes the unique identifier, stored in the SIM, to manage the rights and privileges of each subscriber on various networks.

- Banking - Millions of banking smart cards worldwide are enrolled under the EMV (Europay, MasterCard, VISA) standard. Smart cards have been proven to secure transactions with regularity, so much so that the EMV standard has become the norm.

- Physical access - Basic identity cards for simple authentication of individuals, but also more advanced PIV (Personal Identity Verification) standardized enrollments are present all over the world.

- Healthcare - Smart cards address challenges connected with healthcare data with secure, mobile storage and distribution of patient information, from emergency data to benefits status. Many countries have already adopted smart cards as credentials for their health networks and as a means of carrying an immediately retrievable EHR (Electronic Health Record).

- Digital content protection - Information and entertainment being delivered to customers via satellite or cable to the home DVR (Digital Video Recorder) player or cable box or cable-enabled PC. Home delivery of service is encrypted and decrypted via the smart card per subscriber access. Digital video broadcast systems have already adopted smart cards as electronic keys for protection.

- Loyalty cards - Another use of smart cards is stored value, particularly loyalty programs, that track and provide incentives to repeat customers.

For our purposes, we will focus on smart cards with integrated microprocessor, accessed by a contact interface, which are used in telecommunications as SIM cards.

3.2.1 Microprocessor Cards

These cards have on-card dynamic data processing capabilities. Multifunction smart cards allocate card memory into independent sections or files assigned to a specific function or application. Within the card, there is a microprocessor or micro-controller chip that manages this memory allocation and file access. This type of chip is similar to those found inside all personal computers and when implanted in a smart card, manages data in organized file structures, via a COS (Card Operating System).

The two primary types of smart card operating systems are fixed file structure and dynamic application system. As with all smart card types, the selection of a card operating system depends on the application that the card is intended for.

The fixed file structure COS treats the card as a secure computing and storage device. Files and permissions are set in advance by the card issuer. These specific parameters are ideal and economical for a fixed type of card structure and functions that will not change in the near future. An example of this kind of card is a low-cost employee multi-function badge or credential.

The dynamic application COS contains a type of an *Multi-application operating system* and a Java Card virtual machine - an engine that executes Java Card technology applications, also called applets. Because the card operating systems and applications are more separate, updates can be made. An example card is a SIM card for mobile GSM where updates and security are downloaded to the phone and dynamically changed. This type of card deployment assumes that the applications in the field will change in a very short time frame, thus necessitating the need for dynamic expansion of the card as a computing platform [36].

3.2.2 Smart Cards Standards

There is a wide range of standards linked with the smart cards world. There is a set of well known and strictly followed standards issued by ISO (International Organization for Standardization) for smart cards in general, regarding the physical characteristics of the cards. Another entities which issue standards important for smart cards used in telecommunications are ETSI (European Telecommunications Standards Institute), 3GPP (3rd Generation Partnership Project), Simalliance, Global Platform, Open Platform, Oracle (Sun Microsystems) and others. Specifically the most important standards are:

- ISO 7816 - This standard is the most important specification for the lower layers of the smart card. It is a multi-part international standard broken into fourteen parts. The first three parts deal only with contact smart cards and define the various aspects of the card and its interfaces, including the card's physical dimensions, the electrical interface and the communications protocols. The other parts of the standard refer to both types of smart cards (contact and contactless). They define the card logical structure (files and data elements), various commands used by the application programming interface for basic use, application management, biometric verification, cryptographic services and application naming.

- 3GPP 11.11 (51.011) - It defines the interface between the SIM card and the ME (Mobile Equipment) for use during the network operation phase of GSM as well as those aspects of the internal organization of the SIM card which are related to the network operation phase. This is to ensure the interoperability between the SIM card and the ME independently of the respective manufacturers and operators.

- 3GPP 11.14 (51.014) - The document defines the interface between the SIM and the ME, and mandatory ME procedures, specifically for "SIM Application Toolkit".

- ETSI 102.221 - Defines generic commands which could be exchanged between the SIM and a terminal as file manipulation commands, authenticate commands, etc.

- ETSI 102.222 - Defines administrative commands for smart cards as file creation, deletion, activation, etc.

- ETSI 102.224 - Specifies the security mechanisms for a smart card for OTA (Over the Air) manipulation.

- Global Platform 2.x - Defines smart card life cycle management and a secure manipulation with a smart card during its whole life cycle.
- Open Platform 2.x - Structures smart card architecture, defines an entity of Card Manager.
- Java Card 2.x - Defines the programming language used for an environment present on a smart card.

3.2.3 Smart Card File System

According to the ISO 7816 Part 4 there are three categories of files defined:

- Master file (MF)
- Dedicated file (DF)
- Elementary file (EF)

The Master file is a mandatory file for conformance with the standard and represents the root of the file structure. It contains the file control information and allocable memory. Depending on the particular implementation it may have dedicated files and/or elementary files as descendants. A dedicated file has similar properties to the master file and may also have other dedicated files and/or elementary files as descendants. An elementary file is the bottom of any chain from the root MF file and may contain data as well as file control information. An elementary file has no descendants.

Each file is referenced by a two byte identifier which allows the path to any file to be defined from the root directory. The data structure for an elementary file allows four options:

- Linear fixed - Set of records with the same length.
- Linear variable - Set of records with the variable length.
- Cyclic - A cyclic record.
- Transparent - A raw block of data without a record structure.

A file control information for DF and EF files consist of two parts, FCP (File Control Parameters) and FMD (File Management Data). FCP are defined as ASN.1 (Abstract Syntax Notation One) encoded data field that describes the necessary parameters such as file size, file identifier and optionally the file name. It also defines the type of file and the data structure used.

The file management data is also constructed as an ASN.1 object and it contains Inter-Industry or provider specific objects. A typical usage is for definition of an access rights for files.

3.2.4 Communication with a Smart Card

The smart card acts as an ultimately thin server. It always acts as a slave and never takes an initiative of initiating the communication.

A standardized communication with a smart card was introduced by the ISO 7816 standard in the part 3. The standard defines a concept of APDUs (Application Protocol Data Units). An APDU contains the command or response message. There are 4 cases of APDUs, as shown in Table 3.1:

Case	Input data	Output data
1	NO	NO
2	YES	NO
3	NO	YES
4	YES	YES

Table 3.1: Different cases of APDUs.

With respect to the different cases of APDUs, the structure of the data units is shown in Figure 3.4.

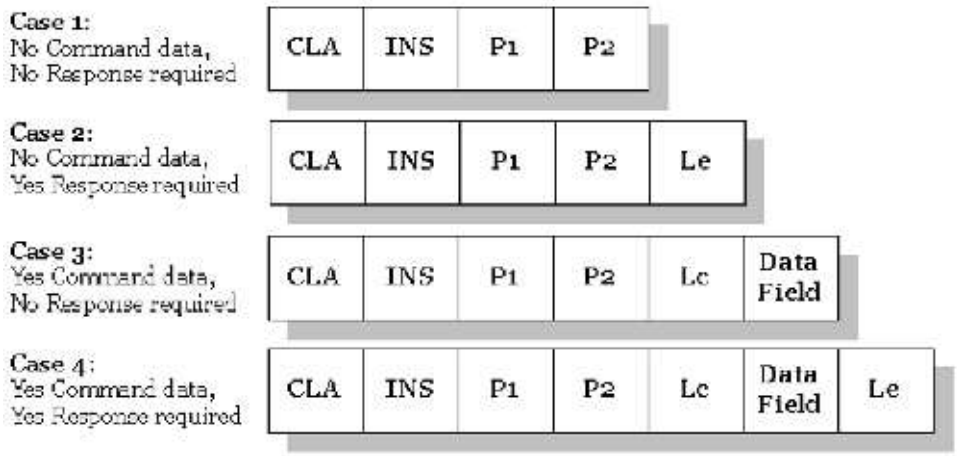


Figure 3.4: APDU command structure.

The response APDU contains the response data field (if present) and two status bytes. The normal response code has been defined as "0x9000" and a number of error conditions have its standardized error codes as well.

3.2.5 Java Card Technology

Java Card technology provides a secure environment for applications that run on smart cards and other devices with very limited memory and processing capabilities. Multiple applications can be deployed on a single card, and new ones can be added to it even after it has been issued to the end user [37].

The Java Card 2.1 standard was released by the Java Card Forum in early 1999. At the same time, ETSI endorsed the use of Java Card in SIM cards and defined the GSM SIM API for Java Cards [38].

The current Java Card platform, as defined by Oracle (Sun Microsystems) and the Java Card Forum, deals with the execution of applets written in the Java Card, which is a subset of Java. Applet management, such as loading, installing or deleting, was deliberately left open to allow proprietary implementations by the vendors. However, the new Global Platform specifications aim to standardize also this process to support full interoperability possibility.

Java Card applets are programs written in the Java Card subset of the Java platform. They make use of the Java Card API, the Java Card virtual machine and the Java Card runtime environment (JCRE) provided by the smart card. Java Card applets are completely different to Java applets executed in a browser or applet viewer, so these two techniques should not be mixed up. The name applet was chosen to distinguish the Java Card applications from other applications in embedded devices, which are normally burnt into ROM, whereas Java Card applets can be dynamically loaded and deleted after the card has been issued.

Unlike other computers, the smart card is only booted once. The life cycle of the virtual machine, runtime environment and all pre-instantiated objects corresponds to the life cycle of the card. Removing the power source only stops the system temporarily, it does not terminate it.

The applet life cycle begins after it has been completely downloaded on to the card, linked with the other applets and registered by the JCRE. It lasts until the end of the card's life cycle or until it is deleted. To initiate the applet life cycle, the JCRE executes the applet install method, which is similar to the main method used in conventional Java: an instance of the applet is created and registered. At the same time, all objects needed by the applet are created and initialised.

Java Card Memory Management

A smart card usually has three types of memory:

- ROM (Read Only Memory) - Read only memory which can be written to only during the card production.
- RAM (Random Access Memory) - Unlike on a "normal" platform, where Java would create an object in RAM memory, in the smart cards environment, the RAM is extremely expensive to provide, and thus usually very limited. This constraint lead to a decision made in Java Card 2.x, that object can be created also outside of the RAM.
- EEPROM (Electrically Erasable Programmable Read-Only Memory) - Non-volatile type of memory in smart cards. Slower than RAM, but its content remains persistent also after a card power supply is removed. Smart cards have physical limitations on number of writes to this memory.

For the reasons stated, Java Card object can be created in a combination of a volatile (RAM) and non-volatile memory (EEPROM). The default behavior is that all objects registered or referenced from a static field become persistent. On the opposite, the transient objects are saved in RAM. There exist two events

of transient objects memory release: COD (Clear on Deselect) and COR (Clear on Reset). Objects in COR are cleared to a default value (null, zero or false) whenever the card is reset or its power supply removed. Objects in COD are cleared to a default value (null, zero or false) each time the applet that created them is deselected, and also when the card is reset or powered down.

Communication between the Terminal and the Applet

Communication with the outside world takes place via the APDU exchange mechanism described earlier. After the system is a powered up or reset, the JCRE enters a loop and waits for any incoming messages (APDUs). If an APDU is received, the JCRE dispatches it and checks whether it is a message for the JCRE itself, such as the command to select or deselect a certain applet. Otherwise, it forwards the message directly to the selected applet. The selected applet must process the APDU and send an answer to the terminal, following which it returns control to the JCRE.

Applets are identified by the applet identifier (AID). On a card, the AID must be unique for each applet. The AID is defined before the applet is loaded to the card. The format of AIDs is specified by ISO 7816 part 5, and in particular for GSM cards, by ETSI 101.220.

3.3 Remote Method Invocation

Remote method invocation (RMI) is a general approach of executing some computations remotely. In Java environments, it is considered to be an object oriented version of the remote procedure calls (RPC) [39].

Remote procedure calls are a kind of an inter-process communication, that allows a program to cause a subroutine to be executed in another address space, which is usually physically on another device, without the programmer explicitly coding the details for this remote interaction.

An RPC is initiated by the client, which sends a request message to a known remote server to execute a specified procedure with supplied parameters. The remote server sends a response to the client, and the application continues its process. There are many variations and subtleties in various implementations, resulting in a variety of different (incompatible) RPC protocols.

The basic building blocks of the remote technology are the Stubs and Skeletons approach. A Stub is located on the client side while Skeleton is located on the server side. The role of the Stub and the Skeleton is to do marshalling and unmarshalling, meaning that it flattens the arguments and return value of a method to be in a standard format for transferring over the network on one side and it builds the same arguments and a return value on the other side. Figure 3.5 shows how RMI communication is performed.

The original Java RMI depends on JVM (Java Virtual Machine) class representation mechanisms and it thus only supports making calls from one JVM to another. The protocol underlying this Java-only implementation is known as Java Remote Method Protocol (JRMP) [40].

As the need to interact with other than Java environments evolved, a CORBA (Common Object Request Broker Architecture) platform was developed. The

CORBA is a standard defined by the Object Management Group that enables software components written in multiple computer languages and running on multiple computers to work together. CORBA uses an interface definition language (IDL) to specify the interfaces which objects present to the outer world. CORBA then specifies a mapping from IDL to a specific implementation language like C++ or Java. Standard mappings exist for Ada, C, C++, Lisp, Ruby, Smalltalk, Java, COBOL, PL/I, Python and others [41].

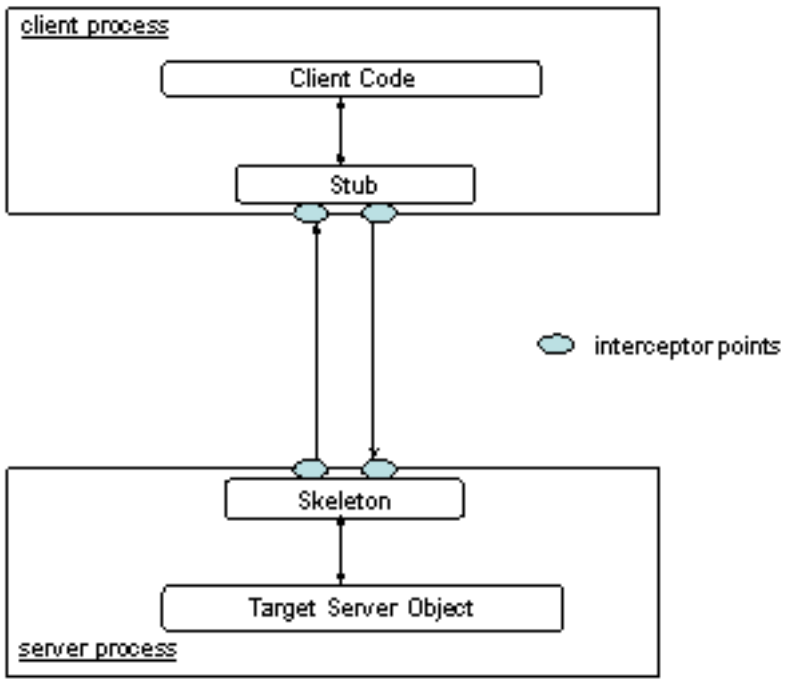


Figure 3.5: Remote Method Invocation communication.

3.3.1 Benefits of Usage

The benefits of using the remote method invocation are most visible when used in distributed computing. Distributed computing involves the design and implementation of applications as a set of cooperating software entities (processes, threads, objects) that are distributed across a network of machines. Each entity has some advantage and offers its services to others transparently over the RMI.

The general advantages include:

- Performance
- Scalability
- Resource sharing
- Fault tolerance

The main issues of the traditional (network based) distributed systems are latency, synchronization and error recovery. Network latency is extended by a RMI

middleware processing overhead. Synchronization issues may be present if the number of entities in the distributed system is high and the network or the nodes are unreliable. Error recovery may be an issue in a case, when a dead node is present in the system, but other nodes do not know it. However, the modern distributed systems copes with those problems and usually its possible to set up specifically designed and adjusted distributed system RMI solution.

3.4 One Time Passwords Algorithms

In the Chapter 1, we have outlined the advantages of One Time Passwords. In this section, we overview the two most common standardized implementations of OTP algorithms.

3.4.1 HOTP

The HMAC (Hashed Message Authentication Code) [9] One-Time Password Algorithm is based on an increasing counter value and a static symmetric key known only to the token and the validation service. According to the RFC 4226, the algorithm was created with respect to the following requirements:

- The algorithm must be sequence or counter based.
- The algorithm should be economical to implement in hardware by minimizing requirements on battery, number of buttons, computational horsepower, and size of LCD display.
- The algorithm must work with tokens that do not support any numeric input, but may also be used with more sophisticated devices such as secure PIN-pads.
- The value displayed on the token must be easily read and entered by the user, which requires the HOTP value to be of reasonable length. The HOTP value must be at least a 6-digit value. It is also desirable that the HOTP value be 'numeric only' so that it can be easily entered on restricted devices such as phones.
- There must be user-friendly mechanisms available to resynchronize the counter.
- The algorithm must use a strong shared secret. The length of the shared secret must be at least 128 bits. The RFC 4226 recommends a shared secret length of 160 bits.

The algorithm and the protocol uses several parameters:

K is a shared secret between client and server. Each HOTP generator has a different and unique secret K.

C is an 8-byte counter value, the moving factor. This counter must be synchronized between the HOTP generator (client) and the HOTP validator (server).

T is a throttling parameter. The server will refuse connections from a user after T unsuccessful authentication attempts.

S is a resynchronization parameter. The server will attempt to verify a received authenticator across S consecutive counter values.

D is a number of digits in an HOTP value.

In order to create the HOTP value the HMAC-SHA-1 algorithm is used [11]. The output of the HMAC-SHA-1 calculation is 160 bits, so it must be truncated to something that can be easily entered by a user. The HOTP value is calculated using the following formula:

$$HOTP(K, C) = Truncate(HMAC - SHA - 1(K, C))$$

The prove, which demonstrates that the best possible attack against the HOTP function is the brute force attack is described in the Appendix A of the RFC 4226.

The exchange of the HOTP values sent from the client to the server needs to be synchronized. However, the validation server needs to cope with situations, where client's counter value was incremented, but not sent to the server. If the value received by the server does not match the value calculated by the client, the server initiates the resynchronization protocol (look-ahead window of size S) before it asks for another OTP value. This parameter S must be as low as possible with user friendliness of the protocol in mind, to avoid the probability of an adversary simply guessing the OTP value as in case of of a high S parameter and short D parameter, it can lead to a serious security vulnerability.

The validation server also needs to detect a possible brute force attack and stop it before it can succeed. One way is to introduce the throttling parameter T, which defines the maximum number of possible attempts for One-Time Password validation. Another option would be to implement a delay scheme, which would increase a delay between the consecutive invalid OTP requests.

3.4.2 TOTP

The Time-Based One-Time Password Algorithm is an extension to the previously described algorithm supporting the time-based moving factor. The HOTP algorithm specifies an event-based algorithm, where the moving factor is an event counter, while the TOTP bases the moving factor on a time value. A time-based variant of the OTP algorithm provides short-lived OTP values, which are desirable for enhanced security. The RFC 6238 summarizes the requirements taken into account for designing the TOTP algorithm.

- The prover and the verifier must know or be able to derive the current Unix time for OTP generation.
- The prover and verifier must either share the same secret or the knowledge of a secret transformation to generate a shared secret.
- The algorithm must use HOTP as a key building block.

- The prover and verifier must use the same time-step value X.
- There must be a unique secret for each prover.
- The keys should be randomly generated or derived using key derivation algorithms.
- The keys may be stored in a tamper-resistant device and should be protected against unauthorized access and usage.

Parameters used in the algorithm:

X represents the time step in seconds.

T0 is the Unix time when we start counting the time steps X.

T1 is the current Unix time.

The TOTP value is calculated using the following formula:

$$TOTP(K,T) = HOTP(K,T)$$
$$where\ T = \lfloor \frac{(T1 - T0)}{X} \rfloor$$

The values of the system parameters X and $T0$ are pre-established during the provisioning process and communicated between prover and verifier as a part of the provisioning step. The security considerations depend on the properties of the underlying HOTP algorithm. Similarly to the underlying building block, the settings of the validation service affects the usability and the security of the protocol. As the validation service cannot know the exact time stamp, when the client's OTP was generated, it may only calculate with a time stamp, when the OTP was received, a look ahead time step window must be introduced. Due to a user and network latency, at least one next time step TOTP value shall be validated. However, longer time-step and lager look ahead window means more time for an adversary to misuse the TOTP value. The RFC 6238 recommends a default time-step size of 30 seconds and only one time stamp look ahead window as a balance between security and usability.

Chapter 4

Analysis

In this chapter, we investigate the existing concepts of the AOSP (Android Open Source Project) which are related to SIM card communication with hardware and analyse possible modifications, which would allow the SIM card RMI to be implemented.

The initial idea of an Android handset communication with a SIM card was, that the Android OS offers an interface of useful commands, such as `boolean verifyChv(byte[] baChv, bChvType)` (verifies PIN code), which could be called from the top level applications. The idea was, that the implementation of the interface is in the Android OS library and it simply translates the interface calls into the APDU messages which are subsequently sent to / received from a Linux kernel driver, which communicates directly with a SIM card as shown in Listing 4.1.

```
/**
 * Verifies CHV (Card Holder Verification)
 * which is known as PIN (Personal Identification Number)
 * for CHV type 1 and as a PUK (Personal Unblocking Code)
 * for CHV type 2.
 *
 * baChv must be supplied as 8 bytes of GSM 8 bits Hexa
 *       Unpacked encoded byte array padded with 0xFF.
 *       (i.e. "1234" in ASCI -> "31323334FFFFFFFF" encoded).
 */
boolean verifyChv(byte[] baChv, byte bChvType) {
        if (baChv.length != 0x08)
                return false;

        if (bChvType != 0x01 || bChvType != 0x02)
                return false;

        byte bClass = 0xA0;
        byte bIns = 0x20;
        byte bP1 = 0x00;
        byte bP2 = bChvType;
        byte bLc = 0x08;
```

```
        short sSW = LinuxSimCardDriver.sendAPDU(
                bClass, bIns, bP1, bP2, bLc, baChv);

        if (sSW == 0x9000)
                return true;

        return false;
}
```

Listing 4.1: PIN code verification method.

This approach would mean that to access an arbitrary SIM card application, we would just need to implement a generic APDU command proxy into the Android OS library by which we could send APDUs from the upper layers of the Android.

However, after the inspection of the source codes of the AOSP, we came to the conclusion, that for the communication with the SIM card, another implementation is used.

4.1 Communication with a SIM Card

Android applications communicate with a SIM card through a telephony stack. The telephony architecture is split between Java (Application Framework part of AOSP) and native code (Libraries) (see Figure 4.1).

The `android.telephony` [43] package exposes all the SIM card capabilities to the other components inside the android framework. This class interfaces with the radio interface layer (RIL) in order to communicate to the baseband radio modem with the help of `com.android.internal.telephony.RIL`. The top-down communication between these layers is done by asynchronous function calls passing an `android.os.Message` [44] instance to be used in order to send the response back to the function caller with the function result within the message itself.

The `com.android.internal.telephony.RIL` class has two internal classes responsible for sending the requests and receiving the responses to and from the RIL daemon (RILD). Both `RILSender` and `RILReceiver` classes run on its own threads interacting with the RILD through a Linux socket to send and receive messages to the baseband radio.

As a container for messages, an `anroid.os.Parcel` [45] is used. A Parcel contains flattened data that are unflattened on the other side of the IPC.

The RIL daemon, which is a native Linux process, loads a proprietary vendor RIL library and registers its radio specific functions implementation into the telephony stack. The RILD receives requests through a Linux socket and processes the request calling the proprietary library's radio function implementation passing the appropriate parameters. The proprietary library returns a response to the telephony stack through a callback function which marshals the response and sends it back to the Java API through the same socket used to receive the request. The Java layers process the request on the RILReceiver class and forwards the response to the original request owner [46].

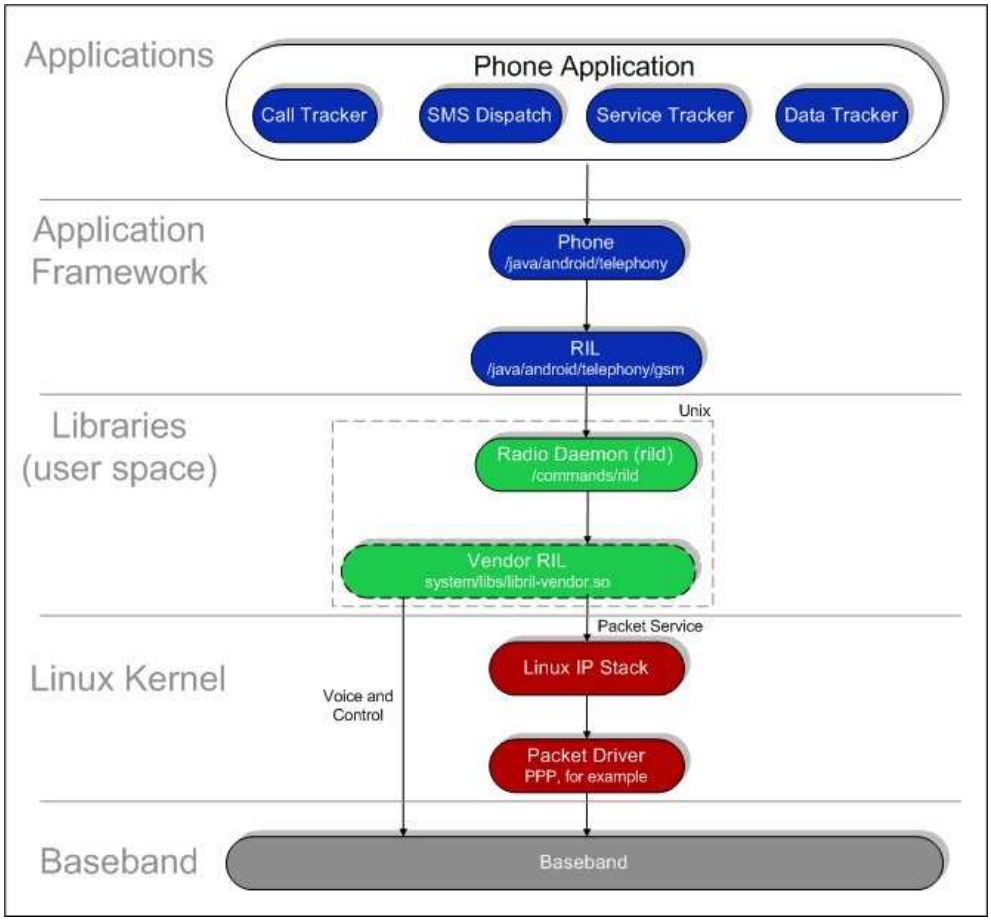


Figure 4.1: Telephony stack [42].

Android initializes the telephony stack and the Vendor RIL at startup as described below:

- RIL daemon reads `rild.lib` path and `rild.libargs` system properties to determine the Vendor RIL library to use and any initialization arguments to provide to the Vendor RIL.
- RIL daemon loads the Vendor RIL library and calls `RIL_Init` to initialize the RIL and obtain a reference to RIL functions.
- RIL daemon calls `RIL_register` on the Android telephony stack, providing a reference to the Vendor RIL functions.

There are two types of communication that the RIL handles:

- Solicited commands - Commands that originates from the RIL daemon, such as SIM PIN, outgoing sms, call handling and others. An example of a solicited command is shown in Figure 4.2.
- Unsolicited commands - Unsolicited responses that originate from the baseband, such as incoming sms, network status change and others. An example of an unsolicited command is shown in Figure 4.3.

The next section describes an implementation of exposed SIM card interface layer through the baseband modem of the handset.

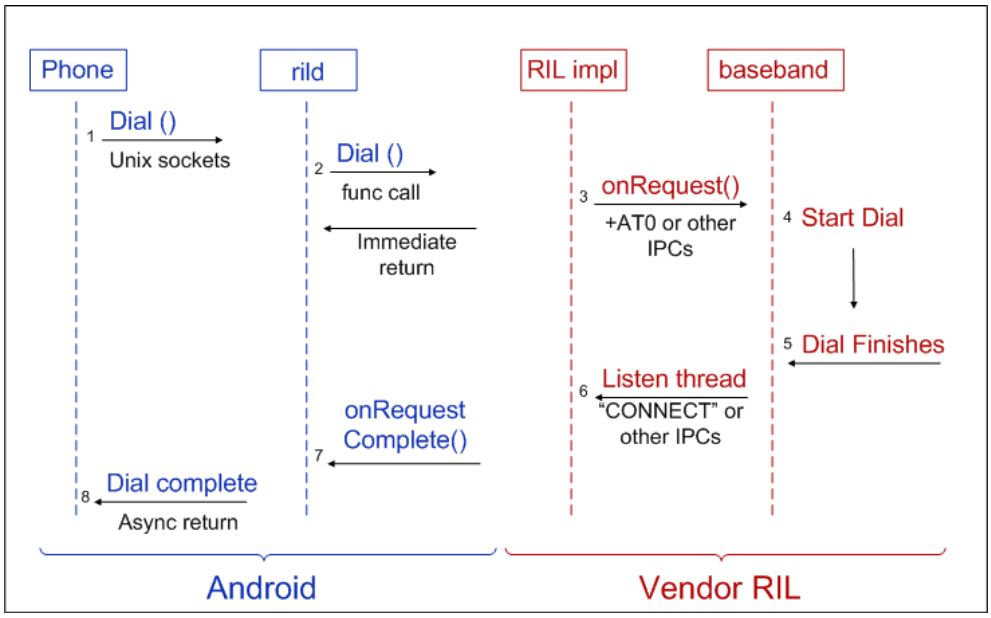


Figure 4.2: Solicited command example [42].

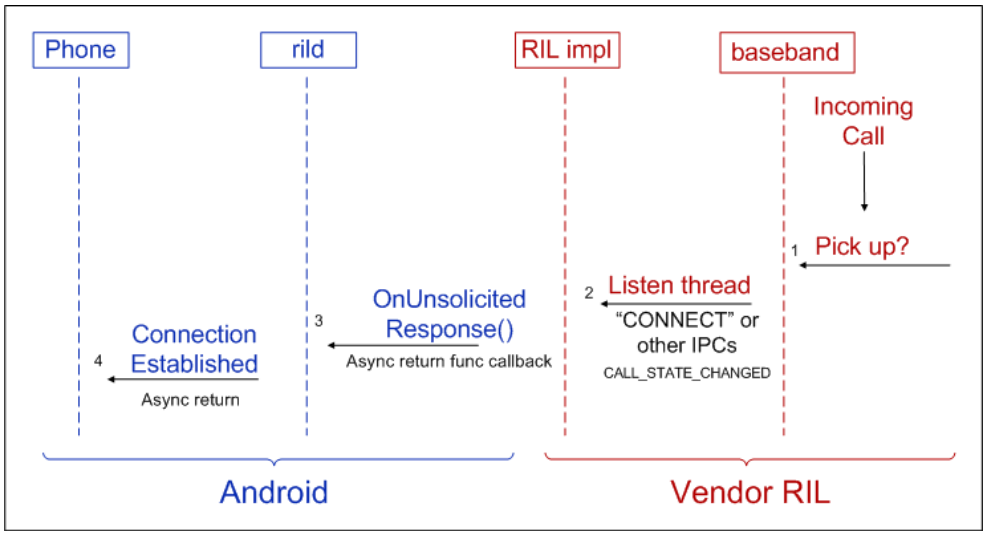


Figure 4.3: Unsolicited command example [42].

4.2 Handsets Implementation

Our initial target was to set up a RMI environment between the SIM card and the handset, HTC Wildfire S A510e, which we had available. However, after the performed research, we realized, that it will not be possible. This section describes the obstacles, which appeared to be impassable in implementing the RIL layer for this handset.

4.2.1 Hardware Structure

Historically, the GSM phones ran a GSM protocol stack (the software implementing the GSM protocol) as well as the user interface and all applications on a single processor, which was called baseband processor [50].

Along with the GSM phones, another market industry was created, producing PDAs (Personal Digital Assistant), which had a wide application base and were initially used as a schedulers, diaries, dictionaries, calendars.

Evolution led to a phase, when it was possible to put both, the GSM phone and the PDA into one case. A smartphone was born.

However, with a reference to historical reasons, the smartphones kept a dedicated processor for the GSM protocol stack (Baseband processor), and another, potentially multi-core, general purpose processor (Application processor) for the user interface and applications.

Due to a market pressure for even smaller phones with even more functions, the industry has produced highly integrated products, uniting the Application processor and the Baseband processor inside one physical package. This approach is called SoC (System on a Chip).

These SoCs have multiple processing cores but unlike the contemporary processor chips, these multiple cores are not available in the OS to run applications with symmetric multiprocessing properties, there is only one core to run the OS and user applications (Application processor).

Generally SoCs chips have the following 4 cores [49]:

- Applications processor - running the OS (Android)
- Applications DSP (Digital Signal Processing) - encoding/decoding of the media
- Baseband processor - running a real-time OS and the GSM stack
- Baseband DSP - encoding/decoding for telephony services

Since the SIM card is connected to the Baseband processor and the OS is running on Application processor, each with their own memory address space, the Android application - SIM card communication must be done via a form of an IPC. For historic reasons, AT commands (messages passing) were chosen [50].

4.2.2 Communication with the Baseband Modem

There are several methods for invoking and controlling modem services. The two most common are through the AT commands and / or through Remote procedural calls.

The AT command set, or so called the Hayes command set, is a specific command language, which consists of a series of short text strings, which combine together to produce complete commands for modem operations. Most of the modems follow the specifications of the Hayes command set.

The AT commands method of inter process communication is by far the most popular also in the smartphones chips, in which AT commands set can be categorized as follows [51]:

- Call Control - Initiation, termination and overall control of the calls.
- Data Call Control - Quality of Service data flow commands.

- Network Service - Commands for Supplementary services, operator selection, locking and registration to the network.
- SMS Control - Commands for sending, notifying, setting SMS services.
- Control & Status - SIM card phone-book access, power management and others.

There is no public documentation of the proprietary vendors RIL implementations. As a result, when one is flashing a device with a custom version of Android, it is required to extract the original vendor RIL library (libVENDOR_ril.so) before the original Android image is replaced, so that a specific proprietary version of the library designed specifically for the baseband chip version is not lost.

Although there is no documentation for vendors baseband interfaces, there is a 3GPP 27.007 standard, which focuses on AT command set for UE (User Equipment) and is "somehow" supported by most vendors. By "somehow" we mean, that the vendors often append vendor specific suffix or prefix to a standardized command. The interested community of experienced users then sniff the baseband in order to gain the knowledge of the AT command set used [52].

A trace of AT commands communication with a baseband is shown in Figure A.1.

However, the 3GPP 27.007 defines commands as mandatory, conditionally mandatory and others as optional. We pick a few AT commands for the readers orientation:

- AT+CPIN - PIN code entry.
- AT+CREG - Network registration.
- AT+CCHO - Open Logical Channel to a SIM card.
- AT+CCHC - Close Logical Channel to a SIM card.
- AT+CSIM - Generic SIM access. With this command, one can send an arbitrary APDU command to the SIM card. This command is marked as optional in the standard.

4.2.3 Hardware Limitations

The handset, which we have tested, HTC Wildfire S 510e, has a CPU model Qualcomm MSM7227 [48]. It is one of the Qualcomms MSM7200 series chip with a SoC architecture, which were designed especially for smartphones.

After we have gained root access to the phone [47], we were able to send AT commands to the devices baseband modem via a `/dev/smd0` terminal. Before we could do that, we had to stop the `rild` process (by `$ stop ril-daemon` or by modifying the init scripts if a fresh communication shall be performed) to stop the interference.

Since the documentation to the Qualcomm's baseband modem interface is not public, we tried to follow the 3GPP 27.007 standard in order to communicate with a SIM card.

We tried to exchange a basic APDU (select Master file) with a SIM card, using a standardized AT+CSIM AT command. As shown in Figure A.2, the baseband modem returned error state (4), the same, as when we sent a random AT+DUMMYTEXT, which led us to a conclusion, that it is an unknown command for this baseband modem.

4.2.4 Vendor Specific RIL Android Implementation

In order to ease the integration of a specific baseband modem provided by a vendor into the Andoird, a reference `ril.c` has been provided and it's stored under `/hardware/ril/reference-ril` directory.

As we have already described, the AOSP provides support for the radio (and also SIM card) access in the Radio Interface Layer, which acts as the interface between the radio HW and the Java Applicaiton Programming Interface (API). The RIL is divided into the following parts:

- The Java RIL accessible to the upper layers of Android, but with a limited set of commands.
- The RIL Daemon acting as an interface between AOSP and the Vendor RIL.
- The Vendor RIL, which is a closed-source and HW specific implementation.
- The Vendor baseband modem HW with an real time OS (for Qulacomm modems it is an OS called REX [53]).

Thus the job of the vendor RIL is to translate all the telephony requests from the Android telephony framework and map them to the corresponding AT commands to the modem, and back again. The mapping is done through `#define TAG MAPPING` in a `/hardware/ril/include/telephony/ril.h` file.

4.3 Android Emulator

As the intention to implement the SIM card - handset RMI into an Android OS, which could be then loaded into a real available device appeared to be unrealistic, we turned our attention to the Android emulator.

The Android Open Source Project contains a QEMU (Quick EMUlator) in its source codes. The binary version of the QEMU emulator is a part of Android SDK (Software Development Kit) and is intended to be used as a test platform for developers [54]. It degrades the need of a physical device to be used, when developing Android applications.

An Android Virtual Device (AVD) is an emulator configuration that enables modeling an actual device by defining hardware and software options to be emulated by the Android Emulator.

An AVD consists of [55]:

- Hardware profile - Defines the hardware features of the virtual device as a camera support, memory size, CPU speed and others.

- Mapping to a system image - Usually used with official releases of the Android OS images. However, also a custom system image can be used.
- Dedicated storage area on hosting device - Emulated user data (and optionally the SD card data) such as settings, applications and others are stored on a hosting device.
- Other options - Other options as a display resolution, skins and others.

4.3.1 QEMU

QEMU is a generic and open source machine emulator and virtualizer [56].

When used as a machine emulator, QEMU can run Operating Systems and programs made for one processor (in case of Android it is most often an ARM processor) on a different processor (i.e. x86). It uses a dynamic binary translation to achieve reasonable speed while being easy to port to new host CPU architectures.

QEMU is able to emulate a full computer system, including peripherals.

When used as a virtualizer, QEMU achieves near native performances by executing the guest code directly on the host CPU. QEMU supports virtualization when executing under the Xen hypervisor or using the KVM kernel module in Linux.

The AOSP contains the QEMU sources in `/external/qemu`. In the `telephony` folder, a dummy simulated SIM card implementation is stored. This implementation returns hard-coded values on AT commands requests from Android RIL system.

4.3.2 PC/SC

A PC/SC is a shortcut for Personal Computer/Smart Card. It is a specification for the integration of smart cards into the computing environments.

Drivers for mainstream Operating Systems are provided, so it is possible to communicate with a smart card from a computer, using a smart card reader as shown in Figure 4.4.

For Linux, CCID (Chip/Smart Card Interface Devices) protocol drivers are available, which are needed to access USB smart card readers. A PCSCD (PC/SC daemon) is used to dynamically allocate/deallocate reader drivers at runtime and manage connections to the readers. And finally, `libpcsclite` library is used, to connect to the PC/SC daemon from a client application and provide access to the desired reader.

With the help of the described tools, it would be possible, to modify the QEMU emulator included in the AOSP project to use the PC/SC interface instead of the dummy SIM card implementation. The Android emulator would be then connected to a real SIM card via a smart card reader connected to a computer running the Android emulator.

4.4 SIM Card RMI Architecture

Our aim is to implement a remote method invocation for security-critical parts of application code on smart cards, which would act as a secure server.

Figure 4.4: Gemalto GemPC Twin USB card reader with a card inserted.

Due to a fact, that a security is a main issue for us and we do not want to have sensitive data present in Android device memory at any time, we cannot speak about a RMI in terms of object serialization and its migration between the SIM card and the Android device. As a result of this, we aim to execute remote methods from an Android application on a SIM card object, but without its exposal to the Android device memory. Thus, a better term for our aim would be RPC instead of the RMI.

The overall architecture of our solution is shown in Figure 4.5.

On a SIM card, there will be an applet serving an Android application as a secure remote server. It would allow satisfactory granularity in terms, that each Android application, which would need to execute security critical code, would have its own Java Card applet, which is important because of the security, which is discussed later, but it also allows reasonable development process, when there is one applet designated for one purpose.

An Android application Stub will always call one SIM card applet (Skeleton), identified by its AID. As the AID is generated according to corresponding standards and it has to be known to the Android application Stub, it is a part of the interface definition.

4.4.1 RMI Generator

An equivalent of `rmic` compiler should be constructed, which would create a Stub and a Skeleton classes, which would reflect a proposed interface between an Android application and a SIM card applet.

An Android application can then call interfaced methods from a Stub. The application developer do not need to care about the underlying communication specifics, almost all the communication is handled by the underling RMI architecture. The only aspect which a developer needs to handle is a remote exception handling.

On the other side, a SIM card applet developer may be interested in the

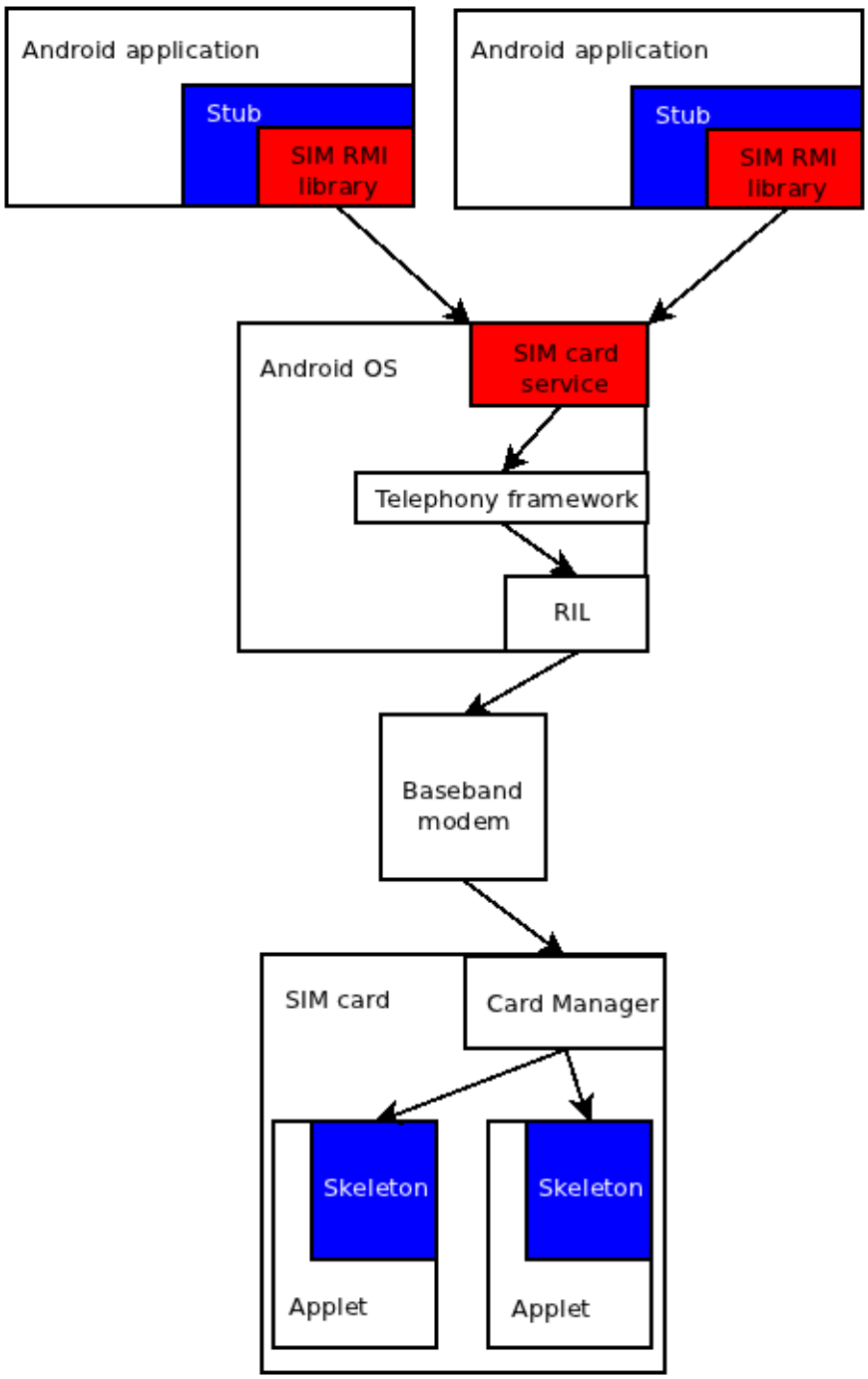


Figure 4.5: Architecture of the proposed RMI environment.

underlying communication. Due to specifics of Java Card development, which were described in the previous chapter, the development on a SIM card side of the RMI implementation must be done very cautiously. As a dynamic memory allocation is highly discouraged, it must be done in advance, when an applet is being installed. This necessity and a fact, that memory size is very limited, usually leads to efforts of reusing allocated memory and thus, one allocated memory block is used for various purposes. The other aspect of this demand is, that a maximum size of transported data is required to be known, prior to the applet deployment.

Because of the mentioned reasons, we have decided, that we will not generate

class files from the interface, but readable Java and Java Card source files will be generated instead. Stub file will be part of an Android application project and will be compiled at the same time as other source files.

On the SIM card side, we generate a Skeleton class, which will implement the interfaced server methods, but it will also be an entry-point class on the applet, implementing the `void process(APDU oApdu)` and `static void install(...)` methods. Empty interfaced methods will be a part of this Skeleton class as well. If memory optimizations are not necessary, a developer can just fill those generated empty method bodies and compile the applet, but in case of interest, optimizations can be done easily.

4.4.2 APDU Interface

Since the only way, how to communicate with a SIM card are the APDUs, a bottom layer interface for the RMI environment will always be APDU based.

In relation to described APDU cases (see Figure 3.4), we can map methods to distinctive Instruction bytes. Class byte will not be used, as its part is used for logical channel selection, so by its usage, we would gain only one nibble of a byte. By mapping remote method calls into a Instruction byte, we would have the upper limit of possible remote methods set to 255 for one applet, which should be satisfactory for every intended purpose. P1 and P2 bytes are thus not required to be mapped either and they should be reserved for future use. Method parameters and return values can be mapped into the data part of the APDUs.

From the limitations of the Java Card platform, only `byte, short, byte[]` data types should be allowed.

Since both sides of the RMI know, in what order the method parameters are, there is no need for TLV (Tag Length Value) notation as the mapping on Stub and Skeleton sides can be done in the same order and the lengths of allowed data types are known as well.

4.4.3 RMI Library

In order to integrate our proposed RMI system, based on a mapping between the methods and the APDUs, we need a layer, which would transport the APDUs generated by the Stub/Skeleton through the Android OS environment to the SIM card.

At least one component of this layer has to be a part of the Android system, because of the Android permissions. There is an option, that this layer would be just signed with a same certificate as was used in the Android system, but since we need to modify other parts of the system image too, and also due to security considerations discussed later, we chose to have this layer to be a part of the Android OS.

4.5 SIM Card RMI Security

One of the main thesis goals was to investigate the execution of security critical parts of application code on smart cards. By excluding all the sensitive data and

all related computation from an Android device memory, we have prevented a possible malicious software installed alongside with an application which requires a certain level of security on a same device, from stealing its secrets.

Despite this, there still exists a threat, that a malicious application would sniff the communication with a SIM card and would be then able to reproduce it, which could lead to secrets compromisation. To prevent this, we need to set up an access control for all SIM card applets, which serve as secure servers, so we can deny a communication with an application, which was not approved to be used with a certain SIM card applet.

4.5.1 Access Control

There are several approaches, how we can achieve the access control for a SIM card applets.

Android Signature Permission

As described in the previous chapter, Android OS offers a possibility of declaring security permissions, which are limiting access to specific components of the system. Lets assume, that an underlying Android OS SIM card access system would require a `signature` permission from Android applications, which would want to use RMI to a smart card. This would mean, that an application requesting access to a smart card, would have to be signed with the same certificate as the underlying Android OS SIM card access system.

This approach would effectively limit applications, which can access the SIM card RMI system, but the disadvantage of the necessity of signing the application by the device OS issuer is clear. On the other hand, for certain situations, where a single entity of Trust is required, this approach might be used.

Android Application Certificates

Android application certificate mechanisms were also described in Chapter 3. These mechanisms can be used as another concept of the access control to the SIM card applets from the Android applications. Lets assume, that there would be a security layer in the underlying Android OS SIM card access system, which would store a list of acceptable Android application signatures with its linkage to the applets AIDs. According to this list, the Android OS RMI system would filter all incoming communication from Android applications and would only allow those connections, which would be stored in the mentioned cross-reference list.

This mapping would guarantee, that only allowed Android applications can communicate with certain SIM card applets, which provide specific security critical tasks.

However, there exists a disadvantage of storing the cross-reference list at one place. As in the previous security proposal, there would have to be a single entity of Trust again, although it would not need to be the same entity, which has signed the Android OS system.

Combined Access Control

We aim to produce a security access scheme, which would allow a truly distributed access control, not reliant on a single entity of Trust, which would be responsible for allowing/denying the connections from Android applications to the SIM card applets.

Global Platform standard is being implemented by more and more smart card manufacturers [57]. Simplified, it allows Service Providers (banks, transport companies, retailers) loading the SIM card applets to SIM cards, which are owned by MNOs (Mobile Network Operators) and are in the field, used by the end users. By introducing the Secure domains on SIM cards a secure separated environment is provided, which allows multiple unrelated providers to have their own isolated content on a SIM card, which can be loaded to a live card via the OTA (Over The Air) protocol.

The access control scheme, which we propose, would be established by a combination of the previous two approaches. More precisely, the Android OS SIM card RMI system would act only as a protocol enforcer and would not act as a decision maker, whether to allow or deny the communication between an Android application and a SIM card applet.

The access control protocol would be enforced in a way, that before each remote request to a SIM card, which has to be made via the Android OS SIM card RMI system, this system sends a request to the selected applet with a predefined structure, asking the applet to return a key, by which the requesting application had to be signed. The SIM card applet can optionally return an empty answer, which would mean, that the service provided by this applet is public. The Android OS RMI system would then check, whether the key returned by a SIM card applet was used to sign the requesting Android application. We do not need to test the application integrity, as this was tested by the Android system at install time of the application.

This allows much more flexible management of Android applications and SIM card applets, as the Android OS SIM card RMI system is only generically implementing an access control protocol, but the decision whether to allow or deny the communication is done by the SIM card applet.

4.6 Link to a Related Work

As the work on this thesis evolved, the initiative, mention in Chapter 2, Secure Element Evaluation Kit for the Android platform has advanced as well.

The initiative solves low-level issues with transport of the APDUs via the Android OS to the SIM card. Since they have identified the issue with the lacking support of the required open baseband modem interface as well, they have implemented an extension of the Android emulator to support a real SIM card connected to a host computer via a PC/SC interface. Although the binary form is not available, the Android OS patches have been released.

The initiative produced an Android service, which is a part of the OS, thus with system privileges which resends received APDUs to the baseband modem, which have been modified by adding a "fake" AT+CSIM command, which is accepted by the modified emulator. The implemented service also tries to restrict

the access to a SIM card by several security features. To use the service, an Android application needs to declare a special permission for this service, so the user is aware, that the application will be accessing a SIM card. This permission based access restriction is effective in order to block malicious software, from accessing a SIM card if the end user is behaving responsibly and is reading/allowing the install pop-up windows. Also the security feature of Android OS, which is showing the permission request at install time has to be enabled in the system.

This restriction would not prevent several installed applications, which all use SIM card, from sniffing each other communication. For those reasons, the authors proposed a more restrictive access scheme based on a single layer cross-reference list, which would check the requesting application certificate and comparing it to the list entries in order to allow it to communicate with certain AIDs.

As we have already discussed, this approach of security restriction is effective, but the granularity and the need of a single entity of Trust is undesirable for us.

The Giesecke & Devrient company, who stands behind this initiative also released the patches of the Android OS, which integrate the APDU transport layer as an Android service, which is a part of the OS.

For the RMI system we proposed, an underlying APDU transport layer is required. Since the SEEK project implementation was made available during our work before it was finished, we have decided to reuse some of its components.

4.7 Goals Revisited

To achieve the goals of the thesis, as stated in Chapter 1, and in accordance with the performed analysis, to build a bottom layer of our solution, the APDU transport component from the SEEK project will be partially reused.

As a security access layer to the sensitive services on a SIM card, we implement a scheme, introduced as a Combined Access Control.

In addition to the APDU and security layers, we will construct a RMI Stub and Skeleton generator (or more accurately the RPC generator), which would serve as a connector between Android applications and Sim card applets.

As a case study implementation, we will implement an OTP generator applet, which will reside in a SIM card. An Android application will be able to generate the OTP from a SIM card with a RMI call, which had been specified in RMI interface, and then show the OTP to the user.

In order to validate the generated One Time Passwords by the SIM card, we will construct a validation server as well.

The performed analysis shows, that it is not possible to implement a RMI system, which could be run on a real available handset device, because of the lacking support of the required baseband modem interface to the SIM card. Handsets which implement the optional parts of the 3GPP standard required for communication with a SIM card in a generic way exist, but they are not accessible to the author. We expect a much better support of this part of the standard in the upcoming years, as the market with NFC enabled handsets is expanding and it is compulsory for such phones to support this feature.

However, we will still be able to integrate the proposed RMI solution for execution of security critical parts of Android applications in a SIM card into the

Android system, which can be run on a modified Android emulator with a SIM card connected to a host computer via the PC/SC interface.

Chapter 5

Implementation

This chapter provides a more detailed insight into our implementation of the RMI system for executing security critical parts of Android applications on SIM cards. The implementation of the proposed case study is described as well.

5.1 RMI Implementation

The RMI system is split into two main parts:

- Android service as a part of the Android OS, containing the access control layer.
- Android library, which is a part of Android application and communicates with the Android RMI service.

The process of developing an Android application, which uses a SIM card remote secure server services via the RMI should begin by defining the interface between the SIM card applet and the Android application.

For definition of the interface, we need a technology independent language, as we will implement the interface in both, Android implementation of Java and in a Java Card language.

For the interface definition, the XML format was chosen. Mainly for its readability, easy parsing and transition to other formats as JSON (JavaScript Object Notation) and also pseudo Java code if required.

5.1.1 Stub and Skeleton Generator

A command line tool written in Java was developed to generate Stub and Skeleton classes from a provided interface file written in the XML format. It takes an interface file as an input and generates Stub and Skeleton files into an output directory.

We have written templates of Stub and Skeleton classes using the XSL (Extensible Stylesheet Language). The target Java and Java Card classes are then generated using the XSLT (XSL Transformations) processor Xalan by Apache.

The XSL is declarative language. It contains rules defining how to transform input XML nodes (the interface file), matching a particular XPath-like pattern.

The mapping between the APDUs and the method calls is done via the Instruction byte as described in Chapter 4. The XSLT rules generate ascending bytes declarations (from 0x01) which are used as a mapping between the exchanged APDUs and the methods.

Interface File

The XML interface file format is defined by the XML Schema file, which is a part of the generator project. It contains required fields, such as the interface name (from which the Stub and Skeleton class names are derived) or remote applet AID. Definition of the public RSA key (its modulus and exponent) is optional and is used for the access control to the applet as described elsewhere. We allow a definition of maximum 254 methods.

Each method has to have a name, return type and parameters defined. Due to the described limitation of Java Card environment, we support 3 primitive types:

- byte
- short
- byte[]

For more complex types, a TLV (Tag Length Value) structure is being used in conjunction with the byte array type, but this is up to the application developers to decide.

In case of the byte array type, we also require a (maximal) length to be defined, due to described memory limits of smart cards.

Stub

Generated Stub class is to be used at Client side, which is in our case an Android application. As discussed earlier, we do not compile the generated classes, so the package name in which the Stub class will be included must be defined after the generation. Otherwise, the generated Stub is ready to be used and do not need any other post-handling.

The `Stub.xsl` template file is using several generic transformation rules, which were moved to `Templates.xsl` as they are used by the Skeleton transformation as well.

The transform rules provide serialization and deserialization of defined interfaced methods into/from the APDUs. The template itself contains static content defining various helper methods. Each method call is wrapped into a channel opening and closing routine of the underlying APDU transfer layer. When opening a channel, the defined AID of the remote SIM card application is used.

Skeleton

Generated Skeleton file is intended to be used as it is, just a package name has to be changed after the generation to make it pass a Java Card compiler. It implements `javacard.framework.Applet` interface, so this Skeleton class will be an entry point for the applet. Empty interfaced methods stubs are generated at

the end of the file and are intended to be filled by the implementor. All the serialization/deserialization in the `void process(APDU oAPDU)` method is generated from the template file `Skeleton.xsl`.

Generated file is well commented and places where the implementor is supposed to fill fields declaration are well marked. This allows also a non experienced Java Card developer to implement required functionality without deeply studying all aspects of Java Card development.

5.1.2 RMI Library

We have implemented a RMI library, which has to be appended to every Android project, which wants to use the RMI functionality from generated Stub and Skeleton classes. The RMI library connects to the underlying APDU transport system and manages data transfers between the Stub and the APDU transport layer. This part of the system reuses parts of the existing SEEK project implementation, especially the parts, which intercept with the underlying Android service, which is a part of the Android OS.

The library main purpose is to establish binding to the remote system service, which serves as an APDU transport layer.

Android Permissions

The library requires a permission (`org.simalliance.openmobileapi.SMARTCARD`) to be granted to the application, which uses the library in order to connect to the underlying system service (APDU transport layer). The underlying system service declares this permission to be of a *Dangerous* type, so during the install, it has to be approved by the user.

5.1.3 APDU Transport Layer

The APDU transport layer is built upon the Android OS version 4.0.3. It contains modifications of the RIL implementation of the main branch of the AOSP to support the AT command for generic SIM card access, which allows us to exchange generic APDUs with the baseband modem (in our case with modified Android emulator, which resends the APDUs to the real SIM card connected to a host computer via the PC/SC interface).

Low level components, as the telephony framework, have been modified in order to support the APDU exchange between the RIL and the upper layers.

As a single entry point for the generic APDU exchange in the Android OS a remote service was created. This service reuses components from the SEEK project implementation version 2.3.2.

This service is installed and registered with a unique Linux UID/GID (`public static final int SMARTCARD_UID = 1028;`), which is always checked by the lower layer components in order to ensure, that only this service can access the APDU exchange capabilities of the lower Android OS layers.

The main interface to this remote service is called `SmartCardService` and is defined in the `ISmartCardService.aidl` file. The whole remote service is a part of the AOSP and is stored under the `packages/apps/SmartCardService` directory.

The interface allows ISO 7816 channel manipulation (selecting the SIM card applets) as well as the actual APDU transmission on a selected channel.

5.1.4 Access Control Layer

The Access control layer is a part of the remote SmartCardSevice. Each time, when an Android application chooses to open a channel to a selected SIM card applet (identified by its AID), the service enforces a security protocol, whether the Android application has a permission to exchange APDUs with the desired applet. If not, a security exception is thrown and the channel is not opened.

The protocol itself is implemented in `AccessController` class. It consists of a request to the desired SIM card applet by a special APDU with Instruction byte set to 0x00. The applet can respond to this instruction in two ways:

- The `publicKeyRsa` element was not declared in the interface XML file - The RMI generator did not append a special security APDU (Instruction byte set to 0x00) handling in the generated Skeleton class `process(APDU oAPDU)` method. The default generated behavior in this case is to return Unknown instruction error status word (0x6D00) to the caller. If the Access control layer receives this status word as an answer to its access check APDU, it considers it as a sign, that the SIM card applet is not providing any sensitive output and the access to it should not be restricted. Thus it allows all APDU communication with this applet until the channel is closed again.

- The `publicKeyRsa` element was properly declared in the interface XML file - The RMI generator generated a static fields within the generated Skeleton class containing the declared public key components (modulus and exponent) and appended an automatic response to the security check APDU (Instruction byte set to 0x00) in the `process(APDU oAPDU)` method, returning static fields with modulus and exponent defined.

If the control layer receives the modulus and exponent from the applet which is about to be selected, it verifies, that the caller application was signed with this key. If yes, it allows the application all APDU exchange with the selected applet, until the channel to this applet is closed. If not, it does not open the channel and throws a security exception.

5.2 Case Study Implementation

As a real-world case study implementation and to demonstrate a practical use case taking advantage of the implemented RMI system, we have constructed a SIM card applet and an Android application, which serves as an interface for generated One Time Passwords by the SIM card applet, where all the sensitive data are stored and where all the computation with the sensitive data is performed. The Android application acts as a client, requesting generation of the OTPs by the remote method calls to the SIM card applet, which serves as a secure remote server.

The main aim of this case study implementation is to allow users with Android handsets to get rid off the hardware tokens as presented in Figure 1.1. The

functionality of our solution as a second factor used within the authentication scheme remains the same, with reasonable security level still ensured.

From the two options of the OTP algorithms (HOTP and TOTP) presented in Chapter 3, we have chosen the HOTP to be implemented.

5.2.1 RMI Interface

We have proposed a simple interface, as only one method is needed to request the OTP generation: `byte[] getOTP()`. We have concluded, that a 4 byte OTP will be generated and presented to the user in hexadecimal notation.

From a generated RSA key-pair, by which the Android application is signed, we have included the public part into the interface file to restrict the access to the SIM card applet, as the generated OTPs will clearly be a sensitive information and access to its generation should be restricted.

5.2.2 Shared Secret

The HOTP algorithm relies on a symmetric 20 bytes long key and on a 8 bytes long synchronized counter.

We keep those two values together in one binary file, where first 20 bytes are the key and the rest 8 bytes consists of the counter on both sides, the SIM card and on the validation server. The key and an initial counter value can be loaded to the SIM card during the card personalization during its production or it can be delivered to a live SIM card in the field via the OTA protocol. In the second case, even the key file transverses the handset, the information is enciphered and the decipher keys are stored only in the SIM card.

5.2.3 SIM Card Applet

The SIM card applet, which uses a generated Skeleton class, implements the RFC 4226 HOTP algorithm. The key is read from the transparent type file as well as the counter value, which is incremented each time the OTP is generated. This approach eases a possible resynchronization correction, as only the binary file contains the resynchronization relevant information.

In opposite to traditional OTP generators, we do not cut the generated 4 byte OTP, neither do we translate the generated OTP into ASCII digits. This approach used to be chosen due to technical constraints, which we do not face, so we can retain even stronger security by using the whole 4 byte long OTP.

5.2.4 Android Application

The Android application contains only one Activity, which offers the user to generate an OTP by pressing a button. The application project contains the generated Stub class and also the SimRMI library. The Activity consists of a logo of the customer, a text box, where the generated OTPs are presented and a button, which generates the OTP.

An instance of the generated remote OTP generator Stub is created in the Activity's `onCreate(Bundle savedInstanceState)` method.

The method called on button pressed event calls the `getOTP()` method on created remote OTP generator instance in a try block and catches eventual Security or generic exceptions. In background, a remote request is sent to the SIM card applet, where the OTP is generated and it is returned back to the Android application, where it is presented on the handset's display.

In the manifest file the `org.simalliance.openmobileapi.SMARTCARD` permission is declared to be used.

5.2.5 Validation Server

A command line Java validation server was developed, to validate the generated OTPs by the SIM card applet. It reads OTP values at its input and outputs either VALID or INVALID strings.

The server uses the same key file as the SIM card applet. It implements a resynchronization window, which is preventing a resynchronization of the SIM card applet and the validation server counters. If the end user generates the OTP on a handset, but does not provide the OTP to the server, the SIM card has a higher counter value than the one, which is being used on the validation server. For those reasons, the validation server, if the presented OTP is not valid with it's counter value, it also tries defined number of consecutive counter values. If there is a match, the validation server's counter is resynchronized. Otherwise, it remains at the same value as before an invalid OTP entry.

Chapter 6

Discussion

In this chapter, we discuss important aspects of the proposed and implemented solution. We elaborate on security issues, handset support, and related work implementation. We also provide a short discussion upon the comparison of differences between the smart card based RMI and the generic, internet based RMI implementation.

6.1 Security Aspects of Implemented Solution

Our RMI solution, enabling the Android applications to easily execute security critical code on a SIM card, gives the application developers an option, how to avoid storing a sensitive data in Android device memory. This protects the secrets from compromisation in case an exploit is found in the Android OS, which would enable a malicious application to overcome its sandbox. We have witnessed several major security flaws both in Android and also in iOS OSs[5] [58] which ensure us, that keeping all sensitive data away from the device's memory and storing them in a special hardware designed for it (smart cards) instead is at least a good habit, if not a must.

In addition to the described potential security threat, we can also model a straightforward attack to the form of the two factor authentication commonly used nowadays, which our solution is immune to.

6.1.1 Threat Model

The advantage of the two factor authentication is the fact, that by verifying what the user knows and what the user has, we can be more sure about the user's true identity.

Many banks worldwide use users' handsets (or more precisely their SIM card subscriptions), as the second factor used for the authentication (what the user has) to theirs internet banking systems. This concept is based on OTPs generated remotely on a bank server and sent to the user via the SMS, when he/she requests a sensitive operation to be performed in the internet banking, such as accessing the account or issuing a payment order. Especially when the user's handset is running Android OS, this approach is not safe.

Consider a situation, when user's computer, which he/she uses for internet banking is compromised, meaning, that the attacker gained full control over this

computer. The two-factor authentication should prevent a situation, when even with compromised computer (login/password to the internet banking are compromised), the second factor (SMS with OTP sent to the user's handset) used within the authentication, prevents the attacker from issuing a payment order to his account.

However, with a full access to user's computer, there is also a high probability, that not only the internet banking credentials were compromised, but also logins/passwords used to access other services were compromised as well. One of them could be the user's Google account, with which he/she accesses not only his/hers Gmail, Picasa or other Google accounts, but also the Google Play portal. Since the user has probably downloaded some applications from the only Android's official application store, his Android device is linked to this compromised account.

The problem comes with a fact, that the Google Play portal allows remote installation of arbitrary application from its database [59]. Surprisingly, the web portal allows the remote installation without any interaction required with the actual linked physical Android device. Neither the permissions required by the remotely installed application has to be approved on the device itself. This allows the attacker to install any application on the victim's device.

If the attacker wants to issue a payment order to his account, he already knows one factor required to authenticate it (the login/password from the compromised computer) and he can install an arbitrary application from Google play to the victim's handset. So he installs a simple SMS sniffing application, which reroutes the incoming OTPs generated by the bank server generator to his device or email. This would allow him to easily break the two factor authentication scheme without the physical possession of the second factor, the Android handset (SIM card's subscription).

Our approach of generating the OTPs locally (yet remotely from the application's view) on a SIM card does not make this attack possible. Even the attacker would remotely install an application, which would want to generate the OTP on a SIM card, our SIM RMI system access layer would not allow the connection to the SIM card by this malicious application, as it could not be signed by the same key as our authentic OTP Android generation application and thus the attempt would be defeated.

6.1.2 Security Concerns

Although we have shown, that our proposed and implemented security scheme is better than the one commonly used, we are aware of situations, when even our model is not satisfactory.

Specifically, the two most hazardous situations, when our security scheme can be compromised are:

- Physical access to the device - With physical access to the Android device with a SIM card, where a security enabled applets reside, it is very easy to compromise the access level in the Android OS by simply removing the SIM card from the device and inserting it into a card reader, where arbitrary APDUs can be exchanged with the SIM card. In our case study implementation, the shared key and the counter value would not be compromised, as

they are protected by SIM card access control schemes, but the generation of the OTPs would be possible by reverse engineering of the APDU protocol used by the Android application.

- Rooted phone - As the whole security scheme of the Android OS APDU transport layer depends on Linux UID/GID checks, for an application with a root access all permissions would be granted. Thus it could easily intercept or modify all communication between the Android applications and the SIM card applets.

6.1.3 Possible Improvements

To clear away all security concerns for Android applications, which require to perform a security critical computation with sensitive data an isolated environment with very restrictive access has to be created. The environment has to be isolated in a way, that even with Android root account or with a physical access to the device, the access to it would still be protected.

The ARM company, which processors are widely used in Android smartphones have introduced a technology called TrustZone [60]. It provides two processors backed by hardware based access control. One processor is used for the Android OS, while the other runs its own secure OS and provides security services to the normal application processor. The ARM refers to those processors (and its componenets) as Normal world and Secure world. Hardware logic present in the TrustZone enabled processor ensures, that Normal world components do not access Secure world resources, enabling construction of a strong perimeter boundary between the two. To switch from the Normal world to the Secure world a PIN code entry is required on a hardware keyboard, which input is directly connected to the Secure world processor. It may not be a special dedicated hardware keypad on a device, but it can also be a normal display keypad, but it is operated by the Secure world processor (Normal world processor does not have access to the display at that time) and users are informed about the Secure state by a dedicated LED diode on a device.

With this approach, the sensitive data are stored and manipulated with only in tamper resistant Secure world, access to which is hardware protected.

6.2 Handsets Support for APDU Transport

The AOSP project allows an arbitrary hardware vendor to implement the Android OS into his hardware. It is up to the manufacturer, whether does he include the support of the APDU transport layer into his device.

The first step of the device manufacturer would be the implementation of the IPC to the baseband modem, which would allow the generic SIM card access by either implementing the 3GPP standardized `AT+CSIM` command or by implementing a proprietary IPC call.

In conjunction to this, the specific RIL implementation has to be provided by the vendor, to allow upper layers of the Android OS to access the RIL methods.

In addition to the radio modifications, the Android service for the APDU transport with its access control scheme has to be a part of the Android OS.

As an evidence, that the manufacturers of the Android handsets are interested in the possibility of enabling the generic APDU access to the SIM card, some of the Sony Xperia series handsets have the SEEK project APDU transport layer already included [61].

6.3 Comparison to Generic RMI

This section provides a short comparison of differences between the smart card based RMI and the generic, internet based Java RMI implementation.

Virtual Machines

Java RMI facilitates object function calls between Java Virtual Machines. This is the first difference when it comes to our smart card RMI solution used for Android applications, as we need to handle method calls between Dalvik Virtual Machines and the Java Card Virtual Machine. Even if we consider the Dalvik Virtual Machine to be the same as the Java Virtual Machine, we would still need to cope with the Java Card VM, which handles much lesser subset of the Java language than the other party of the communication. Thus, we can not speak about function calls between the same Virtual Machines, which effectively prevents us from generically transfer objects from one VM to another, which is not a problem for the Java RMI.

Interface Definition

Another difference comes with the interface definition. Java RMI has incorporated an interface definition, which is a part of the language itself. Our solution of the interface definition is more alike the CORBA's IDL (Interface Definition Language) as it needs to cope with different languages for Stub (Java) and for Skeleton (Java Card). We have chosen an XML format for the interface definition, especially for its easy readability and simple transition to other formats (JSON) if desired.

Another important difference is also the fact, that we do not interface whole remote objects. We interface only remote method calls on a remote object which will be generated from the interface and will serve itself as a single object server implementation, providing the interfaced remote methods.

Stub and Skeleton Generation

The traditional RMI Stub and Skeleton generators, as the `rmic`, create compiled *class* files to be used as Stub and Skeleton. Due to described memory constraints of the Java Card, where dynamic memory allocation is strongly discouraged and where memory buffers sharing, depending on the logic of the applet, is common, we have decided not to compile the generated Skeleton object. Instead, we have generated a readable Java Card source file, containing empty interfaced methods, which are intended to be filled by the implementor.

The generated Stub object is not compiled either and it is generated as a readable Java source file as well.

Remote Objects Handling

The common part of the both RMI systems is, that they both hide the underlying communication between the Stub and the Skeleton, which involves serialization / deserialization, connection initialization, authentication and possibly other tasks.

In opposite to generic, internet based RMI, the smart card RMI clients (Android applications) do not call the registry of the RMI system, as there is only one server (the SIM card) which can be used and it does not advertise its services.

On smart cards, the server part of the RMI does not need to be explicitly started, it is accessible immediately after it is installed and made selectable.

Different Concerns

Traditional, internet enabled RMI systems has to cope with various network problems, as network latency, network outage, routing of the communication and others.

The smart card RMI system does not cope with any of those problems as the connection to the SIM card in the handset is reliable and fast by design.

On the other hand, both RMI systems requires a certain level of security to be ensured. However, the approach of ensuring the secure environment is different.

The internet enabled RMI solution can limit the access to its services by restricting the access only to certain IP addresses and ports. It is also possible to use Java RMI with SSL to protect the transport layer of the connection.

Our proposed smart card RMI system restricts access to its services by enabling the server (SIM card) service to request the clients to be signed by the server provider. This allows the server provider to effectively limit access to its services only to trusted clients.

Chapter 7

Conclusion and Future Work

The thesis goals were to inquire the potential of a remote method invocation in the context of the Android mobile devices and to investigate execution of security-critical parts of application code on smart cards. As a result of the investigation, the real-world case study should have been proposed and implemented.

We have proposed and implemented the RMI system, which enables the Android applications to execute security-critical parts of application code on smart cards. We have compared several approaches of the security mechanisms of the proposed RMI system and we have implemented the one, which emerged to be the most secure and viable.

To demonstrate the benefits of the implemented RMI system, we have proposed and implemented a real-world case study, providing the two factor authentication, which offers several advantages over the existing solutions in terms of security, but also user friendliness.

We have shown, that the integration of the implemented RMI system into the available physical device is not possible due to the lack of support of the required standardized features in the device proprietary hardware interface. Despite this obstacle, we were able to present the results of the thesis in the Android Virtual Device emulated environment with a connection to the real SIM card via the host computer using the PC/SC interface.

The future work on the project consists of two parts. Firstly, to enhance the compliance of the proposed and implemented access scheme of the RMI system with a still evolving Global Platform standard. Secondly, to attempt the integration of the created RMI system into the main branch of the Android OS, which would help to spread the functionality over the newly introduced Android devices.

Bibliography

[1] Dan Morrill: *Announcing the Android 1.0 SDK, release 1*
http://android-developers.blogspot.com/2008/09/announcing-android-10-sdk-release-1.html

[2] Know Your Mobile: *Press Release*
http://www.knowyourmobile.com/glossary/a/446388/android.html

[3] Gartner: *Gartner Says Sales of Mobile Devices in Second Quarter of 2011 Grew 16.5 Percent Year-on-Year; Smartphone Sales Grew 74 Percent*
http://www.gartner.com/it/page.jsp?id=1764714

[4] Secure storage: *Android applications for secure storage of data*
http://www.androidzoom.com/android_applications/secure+storage+on

[5] Android Police: *Massive Security Vulnerability In HTC Android Devices*
http://www.androidpolice.com/2011/10/01/massive-security-vulnerability-in-htc-android-devices-evo-3d-4g-thunderbolt-others-exposes-phone-numbers-gps-sms-emails-addresses-much-more/

[6] JSR177: *Security and Trust Services API for J2ME*
http://java.sun.com/products/satsa/

[7] Using Android securely: *Giesecke & Devrient initiative for more secure Android*
http://www.gi-de.com/en/trends_and_insights/android/android_security/android_security.jsp

[8] UoB: *Token-OTP*
http://www.uob.com.sg/personal/ebanking/microsite/2FA/2fa/p_tokenotp.html

[9] M. Bellare, R. Canetti and H. Krawczyk, "Keyed Hash Functions and Message Authentication", Proceedings of Crypto'96, LNCS Vol. 1109, pp. 1-15.

[10] HOTP: *An HMAC-Based One-Time Password Algorithm*
http://tools.ietf.org/html/rfc4226

[11] HMAC: *Keyed-Hashing for Message Authentication*
http://tools.ietf.org/html/rfc2104

[12] TOTP: *Time-based One-time Password Algorithm*
http://tools.ietf.org/html/draft-mraihi-totp-timebased-00

[13] Giesecke & Devrient
http://www.gi-de.com/en/index.jsp

[14] Giesecke & Devrient: *Secure Flash Solutions*
http://www.gd-sfs.com/the-mobile-security-card/

[15] Smart Card API *Secure Element Evaluation Kit for the Android platform*
http://code.google.com/p/seek-for-android/

[16] Android Architecture: *Major components of the Android operating system*
http://www.android.com/

[17] Open Handset Alliance
http://www.openhandsetalliance.com/

[18] Android Open Source Project: *About*
http://source.android.com/about/index.html

[19] Dalvik: *The Android's Virtual Machine*
http://code.google.com/p/dalvik/

[20] Android: *Basics*
http://developer.android.com/guide/basics/what-is-android.html

[21] Android: *Permissions*
http://developer.android.com/guide/topics/security/permissions.html

[22] Android applications: *Fundamentals*
http://developer.android.com/guide/topics/fundamentals.html

[23] Android Open Source Project: *Android Security Overview*
http://source.android.com/tech/security/index.html

[24] Android Activity: *Lifecycle*
http://www.skill-guru.com/blog/2011/01/13/android-activity-life-cycle

[25] Samsung: *Products RFS Brochure*
http://www.samsung.com/global/business/semiconductor/products/

[26] Android: *Signing Your Applications*
http://developer.android.com/tools/publishing/app-signing.html

[27] Android: *Manifest permission element*
http://developer.android.com/guide/topics/manifest/permission-element.html

[28] O'Reilly Media: *Android Application Development, 1st Edition*
http://developer.android.com/guide/topics/manifest/permission-element.html

[29] Android documentation: *Socket*
http://developer.android.com/reference/java/net/Socket.html

[30] Android documentation: *MessageQueue*
http://developer.android.com/reference/android/os/MessageQueue.html

[31] Android documentation: *Pipe*
http://developer.android.com/reference/java/nio/channels/Pipe.html

[32] Andoird API Guide: *Intents and Intent Filters*
http://developer.android.com/guide/components/intents-filters.html

[33] Wikipedia: *Australia Bank Paypass Card*
http://en.wikipedia.org/wiki/File:Australia_Bank_Paypass_Card.png

[34] Gemalto: *Smart cards basics*
http://www.gemalto.com/companyinfo/smart_cards_basics/what.html

[35] Smart Card Basics: *Overview*
http://www.smartcardbasics.com/smart-card-overview.html

[36] Smart Card Basics: *Types of smart cards*
http://www.smartcardbasics.com/smart-card-types.html

[37] Java Card Technology: *Overview*
http://www.oracle.com/technetwork/java/javacard/overview/index.html

[38] Sim Alliance: *Interoperability Stepping Stones*
Release 2004

[39] Oracle: *Java Remote Method Invocation*
http://www.oracle.com/technetwork/java/javase/tech/index-jsp-138781.html

[40] Oracle: *Remote Method Invocation: Introduction*
http://java.sun.com/developer/onlineTraining/rmi/RMI.html

[41] Oracle: *Introduction to CORBA*
http://java.sun.com/developer/onlineTraining/corba/corba.html

[42] AOSP: *Radio Layer Interface*
http://www.kandroid.org/online-pdk/guide/telephony.html

[43] Android reference: *Telephony*
http://developer.android.com/reference/android/telephony/package-summary.html

[44] Android reference: *Message*
http://developer.android.com/reference/android/os/Message.html

[45] Android reference: *Parcel*
http://developer.android.com/reference/android/os/Parcel.html

[46] David Marques: *Android Telephony Stack*

[47] XDA developers: *How to root Htc Wildfire S*
http://forum.xda-developers.com/showthread.php?t=1702984

[48] Qualcomm developer: *HTC Wildfire S*
https://developer.qualcomm.com/device/htc-wildfire-s

[49] Wikipedia: *MSM7200*
http://en.wikipedia.org/wiki/MSM7000

[50] Harald Welte: *Anatomy of contemporary GSM cellphone hardware*
laforge.gnumonks.org/papers/gsm_phoneanatomylatest.pdf

[51] XDA developers: *How to talk to the Modem with AT commands*
http://forum.xda-developers.com/showthread.php?t=1471241

[52] Fabien Sanglard: *Tracing the baseband*
http://fabiensanglard.net/cellphoneModem/index2.php

[53] Wikipedia: *REX OS*
http://en.wikipedia.org/wiki/REX_OS

[54] Android reference: *Andoird Emulator*
http://developer.android.com/tools/help/emulator.html

[55] Android reference: *Managing Virtual Devices*
http://developer.android.com/tools/devices/index.html

[56] QEMU: *Wiki*
http://wiki.qemu.org/Main_Page

[57] Global Platform: *Full Members*
http://www.globalplatform.org/membershipcurrentfull.asp

[58] Jens Heider, Rachid El Khayari, Fraunhofer SIT: *iOS Keychain Weakness*

[59] Google Play: *About*
https://play.google.com/about/

[60] ARM: *TrustZone*
http://www.arm.com/products/processors/technologies/trustzone.php

[61] SEEK: *Devices*
http://code.google.com/p/seek-for-android/wiki/Devices

Appendix A

Traces

```
[OUT] :AT
[ IN ] :0
[OUT] :ATZE0S0=0Q0R0D3&A1&F1
[ IN ] :0
[OUT] :AT+CMEE=1
[ IN ] :0
[OUT] :AT+CRC=1
[ IN ] :0
[OUT] :AT+CR=1
[ IN ] :0
[OUT] :AT+CREG=1          # Network Registration
[ IN ] :0
[OUT] :AT+FCLASS=0
[ IN ] :0
[OUT] :AT+CMGF=0
[ IN ] :0
[OUT] :AT+CSCS="HEX"
[ IN ] :0
[OUT] :AT+CGREG=1
[ IN ] :0
[OUT] :AT+CUSD=1
[ IN ] :0
[OUT] :AT+CIMI            # What is the SIM Serial number ?
[ IN ] :230 02 75 63 97 856    # 230 = Czech Rep., 02 = O2
```

Figure A.1: RIL communication with a device baseband processor.

```
lubos@laptop:~$ adb shell
root@android:/ # cat /dev/smd0
AT-Command Interpreter ready
+PB_READY
^C
root@android:/ # echo -e 'AT\r' > /dev/smd0
root@android:/ # cat /dev/smd0
0
^C
root@android:/ # echo -e 'AT+CSIM\r' > /dev/smd0
root@android:/ # cat /dev/smd0
4
^C
root@android:/ # echo -e 'AT+CSIM=A0A40000023F00\r' > /dev/smd0
root@android:/ # cat /dev/smd0
4
^C
root@android:/ # echo -e 'AT+CSIM?\r' > /dev/smd0
root@android:/ # cat /dev/smd0
4
^C
root@android:/ # echo -e 'AT+DUMMYTEXT\r' > /dev/smd0
root@android:/ # cat /dev/smd0
4
^C
root@android:/ # echo -e 'AT\r' > /dev/smd0
root@android:/ # cat /dev/smd0
0
^C
```

Figure A.2: Communication with HTC Wildfire S 510e baseband modem (radio version 7.46.35.08).

Appendix B

Content of the Attached CD

The attached CD consists of:

- `bin/` - Executables of the thesis results.
- `etc/` - Configuration files, patch for the AOSP project.
- `src/` - Eclipse archive packages with source codes and documentation to the projects.
- `master_thesis.pdf` - Text of the thesis.
- `README.txt` - Short CD content description.
- `contact.txt` - Contact to the author.

Each directory contains its own `README.txt` which provides a detailed description of the subdirectories content.